Mica Working Design Document
Executive - Part I

Revision 1.0
18–February–1988

These chapters (4 through 8) describe the base components of the Mica executive: The Kernel, Object Architecture, Process Structure, Memory Management, and I/O Architecture.

Restricted Distribution

This information shall not be disclosed to non-Digital Equipment Corporation personnel or generally distributed within Digital Equipment Corporation. Distribution is restricted to persons authorized and designated by the Responsible Engineer or Manager.

This document shall not be left unattended, and, when not in use, shall be stored in a locked storage area.

These restrictions are to be enforced until noted otherwise.

Responsible Engineer/Manager

Document Copy:

Issued by:
DECwest Engineering

digital™
<table>
<thead>
<tr>
<th>Date</th>
<th>Revision Number</th>
<th>Author</th>
<th>Chapter</th>
<th>Summary of Changes</th>
</tr>
</thead>
<tbody>
<tr>
<td>18--February--1988</td>
<td>1.0</td>
<td>Dave Cutler</td>
<td>4, The Kernel</td>
<td>Initial posted version</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Lou Perazzoli</td>
<td>5, Object Architecture</td>
<td>Initial posted version</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Mark Lucovsky</td>
<td>6, Process Structure</td>
<td>Initial posted version</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Lou Perazzoli</td>
<td>7, Memory Management</td>
<td>Initial posted version</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Charles Olivier</td>
<td>8, I/O Architecture</td>
<td>Initial posted version</td>
</tr>
</tbody>
</table>
# TABLE OF CONTENTS

## CHAPTER 4 THE KERNEL

4.1 Overview ......................................................... 4–1
  4.1.1 Requirements ........................................ 4–1
  4.1.2 Functional Description ............................... 4–1
    4.1.2.1 Environment of the Kernel .................. 4–1
    4.1.2.2 Interaction With the Executive ............ 4–2
    4.1.2.3 Primary Kernel Data Structures .......... 4–2
    4.1.2.4 Primary Kernel Functions ................ 4–3
    4.1.2.5 Performance Data Collection .............. 4–4

4.2 System Control Block ......................................... 4–4

4.3 Processor Control Block ..................................... 4–4

4.4 Information Addressed by Processor Index ................. 4–5

4.5 Kernel Use of Hardware Interrupt Priority Levels .... 4–5

4.6 Multiprocessor Synchronization ............................ 4–5
  4.6.1 Executive Multiprocessor Synchronization .... 4–8

4.7 System-Wide Data Structures ................................ 4–9

4.8 Kernel Objects .................................................. 4–10
  4.8.1 Dispatcher Objects .................................. 4–10
    4.8.1.1 Event Object ................................ 4–10
    4.8.1.2 Mutex Object ................................ 4–12
      4.8.1.2.1 Mutex Performance Data .............. 4–14
    4.8.1.3 Queue Object ................................ 4–14
    4.8.1.4 Semaphore Object ........................... 4–15
    4.8.1.5 Timer Object ................................ 4–17
    4.8.1.6 Thread Object ................................ 4–18
      4.8.1.6.1 Thread Accounting Information .... 4–24
    4.8.1.7 Dispatcher Object State Change Synchronization ........ 4–24
      4.8.1.7.1 Wait Block ................................ 4–25
      4.8.1.7.2 Wait ....................................... 4–25
        4.8.1.7.2.1 Wait Test .......................... 4–28
        4.8.1.7.2.2 Wait Satisfy ........................ 4–29
        4.8.1.7.2.3 Unwait ............................... 4–29
      4.8.1.7.3 Wait Performance Data ................ 4–29
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.8.2 Control Objects</td>
<td>4-30</td>
</tr>
<tr>
<td>4.8.2.1 AST Object</td>
<td>4-30</td>
</tr>
<tr>
<td>4.8.2.2 Device Work Queue Object</td>
<td>4-33</td>
</tr>
<tr>
<td>4.8.2.3 Interrupt Object</td>
<td>4-34</td>
</tr>
<tr>
<td>4.8.2.3.1 Interrupt Object Performance Data</td>
<td>4-36</td>
</tr>
<tr>
<td>4.8.2.4 Power-Up Request Object</td>
<td>4-36</td>
</tr>
<tr>
<td>4.8.2.5 Power-Up Status Object</td>
<td>4-37</td>
</tr>
<tr>
<td>4.8.2.6 Process Object</td>
<td>4-38</td>
</tr>
<tr>
<td>4.8.2.6.1 Process Accounting Information</td>
<td>4-40</td>
</tr>
<tr>
<td>4.8.2.7 VAX Port Queue Object</td>
<td>4-41</td>
</tr>
<tr>
<td>4.8.3 Miscellaneous Kernel Routines</td>
<td>4-42</td>
</tr>
<tr>
<td>4.8.3.1 Set Interrupt Priority Level</td>
<td>4-42</td>
</tr>
<tr>
<td>4.8.3.2 Set Event From Interrupt</td>
<td>4-42</td>
</tr>
<tr>
<td>4.9 Scheduling</td>
<td>4-42</td>
</tr>
<tr>
<td>4.9.1 Dispatcher Database</td>
<td>4-43</td>
</tr>
<tr>
<td>4.9.2 Thread Dispatcher</td>
<td>4-44</td>
</tr>
<tr>
<td>4.9.2.1 Thread Becomes Unrunnable</td>
<td>4-45</td>
</tr>
<tr>
<td>4.9.2.1.1 Idle Thread</td>
<td>4-45</td>
</tr>
<tr>
<td>4.9.2.2 Priority of A Thread Changes</td>
<td>4-46</td>
</tr>
<tr>
<td>4.9.2.3 Thread Makes the Transition to Ready State</td>
<td>4-47</td>
</tr>
<tr>
<td>4.9.2.4 Privileged Context Swapping</td>
<td>4-47</td>
</tr>
<tr>
<td>4.9.3 Fork Dispatcher</td>
<td>4-48</td>
</tr>
<tr>
<td>4.9.3.1 Set Event</td>
<td>4-48</td>
</tr>
<tr>
<td>4.9.3.2 Timer Queue</td>
<td>4-48</td>
</tr>
<tr>
<td>4.9.3.3 Quantum End</td>
<td>4-49</td>
</tr>
<tr>
<td>4.9.3.4 Power-Up Request Queue</td>
<td>4-49</td>
</tr>
<tr>
<td>4.9.3.5 Fork Dispatcher Exit</td>
<td>4-50</td>
</tr>
<tr>
<td>4.9.3.6 Fork Queue Performance Data</td>
<td>4-50</td>
</tr>
<tr>
<td>4.9.4 Dispatcher Performance Data</td>
<td>4-50</td>
</tr>
<tr>
<td>4.10 AST Processing</td>
<td>4-50</td>
</tr>
<tr>
<td>4.10.1 Queuing An AST Object</td>
<td>4-51</td>
</tr>
<tr>
<td>4.10.2 AST Delivery Interrupt</td>
<td>4-53</td>
</tr>
<tr>
<td>4.11 Interprocessor Interrupt</td>
<td>4-54</td>
</tr>
<tr>
<td>4.11.1 Interprocessor Interrupt Performance Data</td>
<td>4-56</td>
</tr>
<tr>
<td>4.12 Interval Timer Interrupt</td>
<td>4-56</td>
</tr>
<tr>
<td>4.12.1 System Time</td>
<td>4-57</td>
</tr>
<tr>
<td>4.12.2 Timer Queue Maintenance</td>
<td>4-57</td>
</tr>
<tr>
<td>4.12.3 Quantum End</td>
<td>4-58</td>
</tr>
<tr>
<td>4.12.4 Interval Time Performance Data</td>
<td>4-58</td>
</tr>
<tr>
<td>4.13 Address Space Number (ASN) Management</td>
<td>4-58</td>
</tr>
<tr>
<td>4.14 Powerfail Recovery</td>
<td>4-58</td>
</tr>
<tr>
<td>4.14.1 Driver Power Failure Example</td>
<td>4-59</td>
</tr>
<tr>
<td>4.15 Vector Enable Fault</td>
<td>4-61</td>
</tr>
</tbody>
</table>
CHAPTER 5 OBJECT ARCHITECTURE

5.1 Overview

5.1.1 Introduction

5.1.2 What is an Object?

5.1.3 Scope

5.1.4 Requirements and Goals

5.1.5 Functional Description

5.1.6 Object-Related Operations

5.1.7 Summary of Proposed Changes to Object Architecture

5.2 Functional Interface and Description

5.2.1 When Should an Element in the Executive be an Object?

5.2.2 Object IDs

5.2.3 Object ID Format

5.2.4 Object ID Sequence Numbers

5.2.5 Object Containers

5.2.6 Object Levels

5.2.6.1 System Level

5.2.6.2 System Level Director Mutex

5.2.6.3 Job Level

5.2.6.4 Job Level Director Mutex

5.2.6.5 Process Level

5.2.6.6 Process Level Director Mutex

5.2.7 Container Directory Index Field of the Object ID

5.2.8 Container Directory

5.2.8.1 Object Index Field of the Object ID

5.2.9 Expressibility of Object IDs

5.2.10 Shareability of Object IDs

5.2.11 Translation of an Object ID to an Object Header Address

5.2.12 Container Directory IDs

5.2.13 Object Access By ID Only

5.2.14 Object Service Routines

5.3 Object Type Descriptor (OTD)
5.3.1 Object Type Descriptor Body Format

5.3.1.1 otd$type Field (Static) ........................................ 5–11
5.3.1.2 otd$objhdr_listhead Field (Dynamic) ......................... 5–12
5.3.1.3 otd$count Field (Dynamic) .................................... 5–12
5.3.1.4 otd$dispatcher_object_offset Field ......................... 5–12
5.3.1.5 otd$access_mask Field (Static) ............................... 5–13
5.3.1.6 otd$allocation_listhead_offset Field ....................... 5–13
5.3.1.7 otd$create_disable Field (Dynamic) ......................... 5–13
5.3.1.8 otd$mutex Field ................................................ 5–13
5.3.1.9 otd$allocate Field ............................................. 5–13
5.3.1.10 otd$deallocate Field ......................................... 5–14
5.3.1.11 otd$remove Field ............................................ 5–14
5.3.1.12 otd$delete Field ............................................ 5–15
5.3.1.13 otd$shutdown Field .......................................... 5–15

5.4 Object Header

5.4.1 Object Header Data Structure

5.4.1.1 objhdr$pointer_count Field (Dynamic) ...................... 5–16
5.4.1.2 objhdr$object_id_count Field (Dynamic) .................... 5–17
5.4.1.3 objhdr$type Field (Static) .................................. 5–18
5.4.1.4 objhdr$otd Field (Static) .................................. 5–18
5.4.1.5 objhdr$link Field (Dynamic) ................................. 5–18
5.4.1.6 objhdr$container Field (Dynamic) .......................... 5–18
5.4.1.7 objhdr$level Field (Dynamic) ............................... 5–18
5.4.1.8 objhdr$index Field (Dynamic) ............................... 5–18
5.4.1.9 objhdr$seq_number_low and objhdr$seq_number_high Fields (Dynamic) ................................. 5–19
5.4.1.10 objhdr$dispatcher_object Field (Static) .................. 5–19
5.4.1.11 objhdr$name Field (Dynamic) ............................... 5–19
5.4.1.12 objhdr$owner Field (Static) ................................. 5–19
5.4.1.13 objhdr$acl Field (Dynamic) ................................. 5–19
5.4.1.14 objhdr$allocation_block (Dynamic) ......................... 5–19
5.4.1.15 objhdr$access_mode Field (Static) ........................ 5–20
5.4.1.16 objhdr$transfer_action Field (Static) .................... 5–20
5.4.1.17 objhdr$reference_inhibit Field (Dynamic) ................. 5–20
5.4.1.18 objhdr$temporary_flag Field (Dynamic) ................... 5–20
5.4.1.19 objhdr$temporary_operation Field (Dynamic) ............... 5–20

5.4.2 object_body Record ........................................... 5–20

5.5 Object Container Data Structures

5.5.1 Container Directories as Compared with Object Containers

5.5.1.1 Object Container OTD Remove Procedure ...................... 5–22

5.5.2 Object Container Object Header

5.5.3 Object Container Body

5.5.3.1 con$objtbl Field ............................................. 5–22
5.5.3.2 con$objnamtbl Field (Static or Dynamic, Depending on Implementation) ........................................ 5–23
5.5.3.3 condir$display_index Field (Static) ........................ 5–23
5.5.3.4 objcon$mutex Field (Static) ................................. 5–24
CHAPTER 6 PROCESS STRUCTURE

6.1 Introduction .................................................. 6-1

6.2 Goals/Requirements ........................................... 6-1

6.3 UJPT Hierarchy .................................................. 6-1
  6.3.1 The User Object ........................................... 6-1
    6.3.1.1 Object Structure ...................................... 6-1
      6.3.1.1.1 Security Profile ................................ 6-2
      6.3.1.1.2 Resource Control ................................ 6-2
      6.3.1.1.3 Access Restrictions .............................. 6-3
    6.3.1.2 Functional Interface .................................. 6-4
      6.3.1.2.1 User Creation ..................................... 6-4
      6.3.1.2.2 Get/Set User Information ......................... 6-6
      6.3.1.2.3 User Deletion ..................................... 6-7
  6.3.2 The Job Object ............................................. 6-8
    6.3.2.1 Object Structure ...................................... 6-8
      6.3.2.1.1 Resource Control ................................ 6-9
    6.3.2.2 Functional Interface .................................. 6-9
      6.3.2.2.1 Job Creation ...................................... 6-10
      6.3.2.2.2 Job Deletion ...................................... 6-11
      6.3.2.2.3 Get/Set Job Information .......................... 6-12
  6.3.3 The Process Object ........................................ 6-13
    6.3.3.1 Object Structure ...................................... 6-14
      6.3.3.1.1 Resource Control ................................ 6-15
      6.3.3.1.2 Process Accounting ............................... 6-15
    6.3.3.2 Functional Interface .................................. 6-16
      6.3.3.2.1 Process Creation .................................. 6-17
      6.3.3.2.2 Process Deletion .................................. 6-18
      6.3.3.2.3 Get/Set Process Information ....................... 6-19
    6.3.3.2.4 Process Control Operations ......................... 6-20
      6.3.3.2.4.1 Process Signaling ............................... 6-20
      6.3.3.2.4.2 Process Hibernate/Wake ......................... 6-21
      6.3.3.2.4.3 Process Suspend/Resume ........................ 6-22
  6.3.4 The Thread Object ........................................ 6-23
    6.3.4.1 Object Structure ...................................... 6-23
    6.3.4.2 Functional Interface .................................. 6-25
      6.3.4.2.1 Thread Creation .................................. 6-25
      6.3.4.2.2 Thread Deletion .................................. 6-26
      6.3.4.2.3 Get/Set Thread Information ....................... 6-27
    6.3.4.2.4 Thread Control Operations ......................... 6-29
      6.3.4.2.4.1 Thread Signaling ............................... 6-29
      6.3.4.2.4.2 Thread Hibernate/Wake ........................ 6-29
      6.3.4.2.4.3 Thread Suspend/Resume ......................... 6-30
      6.3.4.2.4.4 Hibernate and Suspend Comparison .............. 6-31
  6.4 UJPT Object Linkages ...................................... 6-31
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.4.1 Linkage Structure</td>
<td>6-32</td>
</tr>
<tr>
<td>6.4.2 Hierarchy Creation</td>
<td>6-32</td>
</tr>
<tr>
<td>6.4.3 Hierarchy Collapse/Deletion</td>
<td>6-33</td>
</tr>
<tr>
<td>6.4.3.1 Force-Exit Routines</td>
<td>6-34</td>
</tr>
<tr>
<td>6.4.3.1.1 User-Object Force-Exit Routine</td>
<td>6-34</td>
</tr>
<tr>
<td>6.4.3.1.2 Job-Object Force-Exit Routine</td>
<td>6-34</td>
</tr>
<tr>
<td>6.4.3.1.3 Process-Object Force-Exit Routine</td>
<td>6-34</td>
</tr>
<tr>
<td>6.4.3.1.4 Thread Object Force Exit Routine</td>
<td>6-34</td>
</tr>
<tr>
<td>6.4.3.1.4.1 Thread Context Entry</td>
<td>6-34</td>
</tr>
<tr>
<td>6.4.3.1.4.2 Thread Exit</td>
<td>6-35</td>
</tr>
<tr>
<td>6.4.3.2 Object Remove Routines</td>
<td>6-35</td>
</tr>
<tr>
<td>6.4.3.2.1 User-Object Remove Routine</td>
<td>6-36</td>
</tr>
<tr>
<td>6.4.3.2.2 Job Object Remove Routine</td>
<td>6-36</td>
</tr>
<tr>
<td>6.4.3.2.3 Process Object Remove Routine</td>
<td>6-36</td>
</tr>
<tr>
<td>6.4.3.2.4 Thread Object Remove Routine</td>
<td>6-36</td>
</tr>
<tr>
<td>6.4.3.3 Object Delete Routines</td>
<td>6-36</td>
</tr>
<tr>
<td>6.4.3.3.1 User-Object Delete Routine</td>
<td>6-37</td>
</tr>
<tr>
<td>6.4.3.3.2 Job Object Delete Routine</td>
<td>6-37</td>
</tr>
<tr>
<td>6.4.3.3.3 Process-Object Delete Routine</td>
<td>6-37</td>
</tr>
<tr>
<td>6.4.3.3.4 Thread-Object Delete Routine</td>
<td>6-37</td>
</tr>
<tr>
<td>6.5 Address Space and Execution Threads</td>
<td>6-37</td>
</tr>
<tr>
<td>6.5.1 Creation</td>
<td>6-37</td>
</tr>
<tr>
<td>6.5.1.1 Initial Thread Creation</td>
<td>6-38</td>
</tr>
<tr>
<td>6.5.1.1.1 Address Space Creation</td>
<td>6-38</td>
</tr>
<tr>
<td>6.5.1.1.2 Execution Thread Creation</td>
<td>6-39</td>
</tr>
<tr>
<td>6.5.1.1.2.1 Address Space Initialization</td>
<td>6-39</td>
</tr>
<tr>
<td>6.5.1.1.2.2 Control Region Initialization</td>
<td>6-40</td>
</tr>
<tr>
<td>6.5.1.1.2.3 Program Image Mapping</td>
<td>6-40</td>
</tr>
<tr>
<td>6.5.1.2 Subsequent Thread Creation</td>
<td>6-40</td>
</tr>
<tr>
<td>6.5.1.2.1 Thread Stack Creation</td>
<td>6-40</td>
</tr>
<tr>
<td>6.5.1.2.2 Control Region Initialization</td>
<td>6-40</td>
</tr>
<tr>
<td>6.5.1.2.3 Transition to new Thread</td>
<td>6-41</td>
</tr>
<tr>
<td>6.5.2 Deletion</td>
<td>6-41</td>
</tr>
<tr>
<td>6.5.2.1 Execution Thread Deletion</td>
<td>6-41</td>
</tr>
<tr>
<td>6.5.2.1.1 In-Context Thread Deletion</td>
<td>6-41</td>
</tr>
<tr>
<td>6.5.2.1.2 Out of Context Thread Deletion</td>
<td>6-41</td>
</tr>
<tr>
<td>6.5.2.2 Address Space Deletion</td>
<td>6-42</td>
</tr>
<tr>
<td>6.6 Exit Status</td>
<td>6-42</td>
</tr>
<tr>
<td>6.6.1 Object Structure</td>
<td>6-42</td>
</tr>
<tr>
<td>6.6.2 Functional Interface</td>
<td>6-43</td>
</tr>
<tr>
<td>6.6.2.1 Exit Status Object Creation</td>
<td>6-43</td>
</tr>
<tr>
<td>6.6.2.2 Get/Set Exit Status Information</td>
<td>6-44</td>
</tr>
<tr>
<td>6.6.3 Usage</td>
<td>6-45</td>
</tr>
<tr>
<td>6.6.3.1 Thread Exit Status Object Usage</td>
<td>6-45</td>
</tr>
<tr>
<td>6.6.3.2 Process Exit Status Object Usage</td>
<td>6-45</td>
</tr>
</tbody>
</table>
6.7 Process/Thread Startup/Rundown Summary .................................................. 6-45
  6.7.1 Startup Summary .................................................................................. 6-46
  6.7.1.1 Additional Thread Startup Summary .............................................. 6-46
  6.7.2 Rundown Summary ............................................................................. 6-47
6.8 System Threads ............................................................................................ 6-50
  6.8.1 System Thread Creation ...................................................................... 6-50
  6.8.2 System Thread Restrictions ................................................................. 6-50
6.9 System Processes ......................................................................................... 6-51
  6.9.1 System Process Creation ..................................................................... 6-51
  6.9.2 System Process Restrictions ................................................................. 6-51
6.10 Mica Quota System Internals ...................................................................... 6-52
  6.10.1 Quota Levels ...................................................................................... 6-52
  6.10.2 Quota Types ...................................................................................... 6-52
  6.10.3 Quota Interfaces ................................................................................. 6-52
    6.10.3.1 Raw Quota Interface .................................................................. 6-52
    6.10.3.2 Pool Quota Interface .................................................................. 6-53

CHAPTER 7 MEMORY MANAGEMENT ................................................................. 7-1
7.1 Overview ...................................................................................................... 7-1
  7.1.1 Requirements ..................................................................................... 7-1
  7.1.2 Functional Description ....................................................................... 7-1
  7.1.2.1 Environment of Memory Management ....................................... 7-1
  7.1.3 Memory Management Data Structures .............................................. 7-2
  7.1.4 Differences from the VAX/VMS Memory Management Subsystem .... 7-2
7.2 Introduction ................................................................................................ 7-4
  7.2.1 Virtual Address Space ......................................................................... 7-4
    7.2.1.1 Virtual Address Format .............................................................. 7-4
  7.2.2 Physical Address Space ....................................................................... 7-4
  7.2.3 Page Table Entries ............................................................................. 7-5
  7.2.4 Software Flags Within the PTE ........................................................... 7-5
    7.2.4.1 Pillar Record Definition for PTE ................................................. 7-5
  7.2.5 Address Translation (from the PRISM SRM) ....................................... 7-6
7.3 Entering Memory Management .................................................................... 7-9
  7.3.1 Execute-Only Protection .................................................................... 7-9
  7.3.2 Nonexecute Protection ........................................................................ 7-9
7.4 Layout of Virtual Address Space ................................................................ 7-11
  7.4.1 User Space .......................................................................................... 7-11
  7.4.2 Shareable Image Space ....................................................................... 7-12
  7.4.3 Control Space ..................................................................................... 7-13
  7.4.4 System Space ..................................................................................... 7-13
    7.4.4.1 Kernel Mode Entry Pages .......................................................... 7-13
    7.4.4.2 Nonpaged System Space ............................................................ 7-13
    7.4.4.3 Paged System Space ................................................................... 7-14
7.4.5 Hyper Space (Outer Space) ........................................ 7-14
7.4.6 Locating the PTE for a Virtual Address ...................... 7-15
  7.4.6.1 Locating the Segment 1 PTE Address .................... 7-16
  7.4.6.2 Locating the Segment 2 PTE Address .................... 7-16
  7.4.6.3 Translating a Virtual Address to Physical Address .... 7-16
  7.4.6.4 Locking Pages in Memory for I/O ....................... 7-16
    7.4.6.4.1 Steps to Lock Pages in Memory .................... 7-17
  7.4.6.5 Unlocking Pages from Memory ............................ 7-18
    7.4.6.5.1 Steps to Unlock Pages from Memory ................ 7-18

7.5 PFN Database .................................................. 7-18
  7.5.1 Pillar Record for PFN Database ............................. 7-19
  7.5.2 PFN Database: FLINK Element .............................. 7-21
  7.5.3 PFN Database: BLINK Element .............................. 7-21
  7.5.4 PFN Database: REFERENCE_PTE Element (Pointer to PTE) .... 7-21
  7.5.5 PFN Database: ORIGINAL_PTE Element ...................... 7-21
  7.5.6 PFN Database: STATE Element .............................. 7-21
  7.5.7 PFN Database: PAGE_TYPE Element .......................... 7-23
  7.5.8 PFN Database: REFERENCE_COUNT Element .................. 7-23
  7.5.9 PFN Database: SHARE_COUNT Element ....................... 7-23
  7.5.10 PFN Database: EVENT_ADDRESS Element ................... 7-23
  7.5.11 Page Lists .............................................. 7-23
    7.5.11.1 Zeroed Page List ................................... 7-24
    7.5.11.2 Free Page List ...................................... 7-24
    7.5.11.3 Standby Page List ................................... 7-24
    7.5.11.4 Modified Page List ................................. 7-25
    7.5.11.5 Bad Page List ...................................... 7-25
  7.5.12 Retrieving a Free Page .................................. 7-25
  7.5.13 Paging Files ............................................ 7-26

7.6 Address Space Data Structures ............................... 7-26
  7.6.1 Process Control Block .................................... 7-26
  7.6.2 Address Space Tables .................................... 7-26
    7.6.2.1 Working Set List .................................... 7-27
  7.6.3 Address Space Numbers ................................... 7-28

7.7 System Services for Mapping Address Space ................. 7-30
  7.7.1 Create Section Service .................................. 7-30
    7.7.1.1 Subsection Descriptor .............................. 7-31
  7.7.2 Map Section Service ..................................... 7-32
    7.7.2.1 Flow of Create Section ............................ 7-33
  7.7.3 Deletion of Mapping Objects, Sections and Segments ...... 7-33
  7.7.4 Prototype PTE ........................................... 7-34
  7.7.5 Locking of Valid Prototype PTEs in Memory .............. 7-34
  7.7.6 Address Space Deletion .................................. 7-35

7.8 The Pager ................................................... 7-35
  7.8.1 PTE Formats for Segment 1 PTEs ........................... 7-36
  7.8.2 PTE Formats for Segment 2 PTEs ........................... 7-37
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>7.8.3 PTE Formats for Prototype PTEs</td>
<td>7-37</td>
</tr>
<tr>
<td>7.8.4 Making PTEs Valid</td>
<td>7-37</td>
</tr>
<tr>
<td>7.8.4.1 Demand Zero PTE</td>
<td>7-38</td>
</tr>
<tr>
<td>7.8.4.2 Subsection Format of PTE</td>
<td>7-38</td>
</tr>
<tr>
<td>7.8.4.3 Page File Offset Format</td>
<td>7-38</td>
</tr>
<tr>
<td>7.8.4.4 Transition Pages</td>
<td>7-38</td>
</tr>
<tr>
<td>7.8.4.4.1 Page in Transition with Write in Progress</td>
<td>7-39</td>
</tr>
<tr>
<td>7.8.4.4.2 Page in Transition with Read in Progress</td>
<td>7-39</td>
</tr>
<tr>
<td>7.8.4.5 Prototype PTE</td>
<td>7-39</td>
</tr>
<tr>
<td>7.8.4.6 Valid Page with Copy On Modify (Fault on Write)</td>
<td>7-39</td>
</tr>
<tr>
<td>7.9 Kernel Mode Entry Pages</td>
<td>7-40</td>
</tr>
<tr>
<td>7.10 Memory Management Paging I/O</td>
<td>7-40</td>
</tr>
<tr>
<td>7.11 Synchronization Techniques</td>
<td>7-41</td>
</tr>
<tr>
<td>7.11.1 Synchronization Techniques for Working Set Lists</td>
<td>7-42</td>
</tr>
<tr>
<td>7.11.2 Synchronization Techniques for PPN Database</td>
<td>7-42</td>
</tr>
<tr>
<td>7.11.3 Synchronization Techniques for Paging Files</td>
<td>7-42</td>
</tr>
<tr>
<td>7.11.4 Translation Buffer Synchronization</td>
<td>7-42</td>
</tr>
<tr>
<td>7.11.5 Instruction Stream Synchronization</td>
<td>7-42</td>
</tr>
<tr>
<td>7.11.6 Page Faults at Elevated IPL</td>
<td>7-43</td>
</tr>
<tr>
<td>7.12 Managing Physical Pages</td>
<td>7-43</td>
</tr>
<tr>
<td>7.12.1 Modified Page Writer</td>
<td>7-43</td>
</tr>
<tr>
<td>7.12.2 Balance Set Management</td>
<td>7-44</td>
</tr>
<tr>
<td>7.12.3 Page Fault Clustering</td>
<td>7-45</td>
</tr>
<tr>
<td>7.13 Zero Page Thread</td>
<td>7-46</td>
</tr>
<tr>
<td>7.13.1 Performance and Accounting Information Collected</td>
<td>7-46</td>
</tr>
<tr>
<td>7.13.1.1 System-wide Information</td>
<td>7-46</td>
</tr>
<tr>
<td>7.13.1.2 Per Process Information</td>
<td>7-47</td>
</tr>
<tr>
<td>7.14 Allocation of Pool from Paged and Nonpaged Pool</td>
<td>7-47</td>
</tr>
<tr>
<td>7.15 Memory Management Executive Routines for I/O support</td>
<td>7-48</td>
</tr>
<tr>
<td>7.16 Memory Management Support for Page Fault Monitoring</td>
<td>7-48</td>
</tr>
<tr>
<td>7.16.1 Contents of Buffer</td>
<td>7-49</td>
</tr>
<tr>
<td>7.16.2 Example of Using Page Fault Monitoring</td>
<td>7-50</td>
</tr>
<tr>
<td>7.17 Memory Management Executive Services</td>
<td>7-51</td>
</tr>
<tr>
<td>7.17.1 Adjust Working Set Limit</td>
<td>7-51</td>
</tr>
<tr>
<td>7.17.2 Create Virtual Address Space</td>
<td>7-51</td>
</tr>
<tr>
<td>7.17.3 Delete Virtual Address Space</td>
<td>7-52</td>
</tr>
<tr>
<td>7.17.4 Expand Stack</td>
<td>7-52</td>
</tr>
<tr>
<td>7.17.5 Lock Pages in Memory</td>
<td>7-52</td>
</tr>
<tr>
<td>7.17.6 Unlock Pages from Memory</td>
<td>7-52</td>
</tr>
<tr>
<td>7.17.7 Lock Page in Working Set</td>
<td>7-53</td>
</tr>
<tr>
<td>7.17.8 Unlock Page from Working Set</td>
<td>7-53</td>
</tr>
<tr>
<td>7.17.9 Set Protection on Pages</td>
<td>7-53</td>
</tr>
</tbody>
</table>
CHAPTER 8 I/O ARCHITECTURE

8.1 Introduction ............................................. 8-1

8.2 Overview .................................................. 8-1
  8.2.1 Function Processors .................................. 8-1
  8.2.2 Objects Used by the I/O System ....................... 8-2
    8.2.2.1 FPU Object ..................................... 8-2
    8.2.2.2 Channel Object .................................. 8-2
    8.2.2.3 FPD Object ..................................... 8-2
  8.2.3 I/O Request Synchronization ......................... 8-2
  8.2.4 I/O Service Routines ................................ 8-3

8.3 Functional Description .................................. 8-4
  8.3.1 System Services, Executive Services, and Procedures .. 8-4
  8.3.2 Function Processors .................................. 8-9
    8.3.2.1 Function Processor Types ......................... 8-9
    8.3.2.2 Interface Classes ................................ 8-9
    8.3.2.3 Function Processor Unit ......................... 8-10
      8.3.2.3.1 FPU States ................................ 8-10
      8.3.2.3.2 FPU Sequence Number ....................... 8-11
      8.3.2.3.3 FPU State Change AST ....................... 8-12
  8.3.3 Requesting I/O ....................................... 8-13
    8.3.3.1 User-Defined Data Structures ..................... 8-13
      8.3.3.1.1 I/O Parameter Record ....................... 8-13
      8.3.3.1.2 I/O Status Block ........................... 8-14
    8.3.3.2 Function Codes Used by exec$requests_io .......... 8-15
      8.3.3.2.1 Get FPU Information Function Code .......... 8-15
      8.3.3.2.2 FPU Access Function Code ................... 8-16
      8.3.3.2.3 Deaccess Function Code ...................... 8-17
    8.3.3.3 Get Channel Information Function code ............ 8-17
      8.3.3.3.1 Set FPU Information Function Code .......... 8-17
    8.3.3.4 Request Synchronization ......................... 8-17
      8.3.3.4.1 Synchronous I/O Requests .................... 8-18
      8.3.3.4.2 Asynchronous I/O Requests ................... 8-18
    8.3.3.5 I/O Request Packets ............................. 8-18
  8.3.4 Function Processor Parameter Processing ............. 8-19
8.3.5 I/O Request Processing
  8.3.5.1 Procedure-Based Processing 8-20
  8.3.5.2 System Thread Processing 8-20
  8.3.5.3 System Threads Driving Hardware Devices 8-21
    8.3.5.3.1 Startup and Initialization 8-21
    8.3.5.3.2 Creating the Thread 8-21
    8.3.5.3.3 Connecting to an Interrupt 8-21
    8.3.5.3.4 Wait and Signal Operations 8-21
  8.3.5.3.5 Request Queue 8-22
  8.3.5.3.6 Events 8-22
  8.3.5.3.7 Mutexes 8-22
  8.3.5.3.8 FPUs 8-23
  8.3.5.3.9 Synchronizing with Interrupt Procedures and Power Fail 8-23
  8.3.6 Function Processor Callback 8-24
    8.3.6.1 Performing Callbacks 8-26
  8.3.7 I/O Completion 8-27
  8.3.8 I/O Cancellation 8-27
  8.3.9 I/O Security 8-28
    8.3.9.1 Protecting FPU objects 8-28
      8.3.9.1.1 FPU-Specific Access Types 8-28
      8.3.9.1.2 Restricting Function Codes Access 8-30
      8.3.9.1.3 Restricting Channel Connections to an FPU 8-30
    8.3.9.2 Protecting Channel Objects 8-30
      8.3.9.2.1 Channel Specific Access Types 8-30
      8.3.9.2.2 Accessing an I/O Object via a Channel 8-31
  8.3.10 I/O System Configuration 8-33
    8.3.10.1 Configuration of Virtual Layers 8-33
    8.3.10.2 Connection of Channels, FPUs, and FPDs 8-34
  8.3.11 Error Logging and Diagnostics 8-34

8.4 Internal Design 8-35
  8.4.1 I/O Data Structures 8-35
    8.4.1.1 The I/O Request Packet Structure 8-35
      8.4.1.1.1 IRP Free Area 8-36
      8.4.1.1.2 Function Processor Parameter Record 8-38
      8.4.1.1.3 Host Transfer List Data Structure 8-39
      8.4.1.1.4 State Change AST Data Structure 8-40
  8.4.2 FPD Functional Interface and Design 8-41
    8.4.2.1 FPD Object Body Structure Definition 8-41
    8.4.2.2 Creating FPD Object 8-42
      8.4.2.2.1 create_fpd 8-42
    8.4.2.3 Configuring a Function Processor 8-43
      8.4.2.3.1 configure_fp 8-43
      8.4.2.3.2 Function Processor's Configure FP Procedure 8-44
    8.4.2.4 Removing and Deleting FPD Objects 8-45
      8.4.2.4.1 remove_fpd 8-45
      8.4.2.4.2 delete_fpd 8-45
8.4.2.5 Unloading FPD ........................................ 8-46
  8.4.2.5.1 e$unload_fpd ...................................... 8-46
  8.4.2.5.2 Function Processor's Unload FPD Procedure .... 8-47
8.4.3 FPU Functional Interface and Design .................. 8-47
  8.4.3.1 FPU Object Structure Definition ................. 8-48
  8.4.3.2 Creating FPU Object ................................ 8-49
    8.4.3.2.1 exec$create_fpu ................................ 8-49
    8.4.3.2.2 Function Processor's Initialize FPU Procedure. 8-51
  8.4.3.3 Binding FPU Objects .................................. 8-53
    8.4.3.3.1 Connecting Channels to a Bound FPU Object ... 8-54
    8.4.3.3.2 e$bind_fpus .................................... 8-57
    8.4.3.3.3 e$unbind_fpus .................................. 8-58
  8.4.3.4 Getting FPU Information ........................... 8-58
    8.4.3.4.1 exec$get_fpu_information ....................... 8-58
    8.4.3.4.2 Function Processor's Get FPU Information Procedure. 8-59
  8.4.3.5 Misc. FPU Service Routines ....................... 8-60
    8.4.3.5.1 e$register_ast_state_change ................... 8-60
    8.4.3.5.2 e$deregister_ast_state_change .................. 8-61
    8.4.3.5.3 e$change_fpu_state ................................ 8-62
    8.4.3.5.4 io$state_change_ast_cleanup ................... 8-63
    8.4.3.5.5 e$verify_io_access ................................ 8-63
    8.4.3.5.6 e$set_fpu_irp_size ................................ 8-64
  8.4.3.6 Removing an FPU ................................... 8-65
    8.4.3.6.1 io$remove_fpu .................................. 8-65
    8.4.3.6.2 Function Processor's Remove FPU Procedure ... 8-66
    8.4.3.6.3 io$delete_fpu ................................... 8-66
    8.4.3.6.4 Function Processor's Delete FPU Procedure ... 8-67
8.4.4 Channel Functional Interface and Design ............. 8-69
  8.4.4.1 Channel Object Structure Definition .............. 8-69
  8.4.4.2 Creating a Channel ................................ 8-72
    8.4.4.2.1 exec$create_channel .............................. 8-73
  8.4.4.3 Removing and Deleting a Channel .................. 8-74
    8.4.4.3.1 io$remove_channel ............................... 8-74
    8.4.4.3.2 io$delete_channel ............................... 8-75
  8.4.4.4 Get Channel Information ........................... 8-75
    8.4.4.4.1 exec$get_channel_information .................... 8-75
  8.4.4.5 Request I/O Description ........................... 8-76
    8.4.4.5.1 exec$request_io .................................. 8-76
    8.4.4.5.2 e$request_io_trusted .................................. 8-77
    8.4.4.5.3 io$initialize_io_parameters .................... 8-80
    8.4.4.5.4 Function Processor's Initialize I/O Parameters Procedure. 8-81
    8.4.4.5.5 e$execute_io ..................................... 8-82
    8.4.4.5.6 Function Processor's Execute I/O Procedure ... 8-83
    8.4.4.5.7 e$asynchronous_io_call .......................... 8-84
    8.4.4.5.8 Function Processor's Synchronous I/O Call Procedure. 8-85
  8.4.4.6 Misc. Channel Services ............................ 8-85
8.4.4.6.1  e$set_callback_table ........................................... 8–85
8.4.4.6.2  e$remove_callback_table ...................................... 8–86
8.4.4.6.3  e$set_channel_access ........................................... 8–86
8.4.4.6.4  e$clear_channel_access ......................................... 8–87
8.4.4.6.5  e$synch_channel_with_fpu ...................................... 8–87

8.4.5  IRP Processing Executive Services .................................. 8–88

8.4.5.1  FP Parameter Record Allocation and Deallocation ......... 8–88
  8.4.5.1.1  e$allocate_fp_parameter_record ............................. 8–88
  8.4.5.1.2  e$deallocate_fp_param_record .............................. 8–89
  8.4.5.1.3  e$extend_fp_parameter_record .............................. 8–90
  8.4.5.1.4  e$allocate_irp .................................................. 8–90
  8.4.5.1.5  e$deallocate_irp .............................................. 8–91
  8.4.5.1.6  e$get_size_of_htl ........................................... 8–91

8.4.6  Synchronizing I/O ....................................................... 8–92
  8.4.6.1  exec$synchronize_io ............................................. 8–92
  8.4.6.2  e$synchronize_io ................................................ 8–93
  8.4.6.2.1  e$synchronize_io_trusted .................................... 8–94

8.4.7  I/O Completion Functional Interface and Design ............. 8–95
  8.4.7.1  e$complete_io .................................................. 8–95
  8.4.7.1.1  Special Kernel-Mode AST Processing ..................... 8–96
     8.4.7.1.1.1  Function Processor's Complete I/O Procedure ...... 8–97

8.4.7.2  Direct I/O Buffers ................................................. 8–99
  8.4.7.2.1  e$lock_io_buffer ............................................ 8–99
  8.4.7.2.2  e$unlock_io_buffers ......................................... 8–99

8.4.7.3  Paging I/O .......................................................... 8–100

8.4.8  Canceling I/O Requests ............................................. 8–100
  8.4.8.1  exec$s cancel_io ............................................... 8–101
     8.4.8.1.1  Function Processor's Cancel I/O Request Procedure 8–102
     8.4.8.1.1.1  Manipulating an IRP on the Request Queue ....... 8–103

8.4.8.2  e$cancel_io_by_thread .......................................... 8–104
  8.4.8.3  e$cancel_io_by_channel ....................................... 8–105
  8.4.8.4  e$insert_cancel_procedure .................................. 8–106
  8.4.8.5  e$remove_cancel_procedure .................................. 8–107

8.4.9  Startup and Initialization of the I/O Architecture ....... 8–107

8.4.10  Function Processor Startup and Initialization Procedure 8–108
     8.4.10.1  FP & FPU Initialization and Creation Overview .... 8–108

8.5  Outstanding Issues ..................................................... 8–109

EXAMPLES

6–1  User Object Structure .............................................. 6–2
6–2  Resource Control Structures ....................................... 6–3
6–3  Access Restriction Data Structures ............................. 6–4
6–4  User Object Creation System Interface .......................... 6–4
6–5  User Record Structure ............................................. 6–6
6–6  Get/Set User Information System Interface .................... 6–6
6–7  User Object Deletion System Interface .......................... 6–8
<table>
<thead>
<tr>
<th>Page Range</th>
<th>Section</th>
</tr>
</thead>
<tbody>
<tr>
<td>6–8</td>
<td>Job Object Structure</td>
</tr>
<tr>
<td>6–9</td>
<td>Job Object Creation System Interface</td>
</tr>
<tr>
<td>6–10</td>
<td>Job Record Structure</td>
</tr>
<tr>
<td>6–11</td>
<td>Job Object Deletion System Interface</td>
</tr>
<tr>
<td>6–12</td>
<td>Get/Set Job Information System Interface</td>
</tr>
<tr>
<td>6–13</td>
<td>Process Object Structure</td>
</tr>
<tr>
<td>6–14</td>
<td>Process Accounting Structure</td>
</tr>
<tr>
<td>6–15</td>
<td>Process Object Creation System Interface</td>
</tr>
<tr>
<td>6–16</td>
<td>Process Record Structure</td>
</tr>
<tr>
<td>6–17</td>
<td>Process Object Deletion System Interface</td>
</tr>
<tr>
<td>6–18</td>
<td>Get/Set Process Information System Interface</td>
</tr>
<tr>
<td>6–19</td>
<td>Signal Process System Interface</td>
</tr>
<tr>
<td>6–20</td>
<td>Hibernate/Wake Process System Interface</td>
</tr>
<tr>
<td>6–21</td>
<td>Suspend/Resume Process System Interface</td>
</tr>
<tr>
<td>6–22</td>
<td>Thread Object Structure</td>
</tr>
<tr>
<td>6–23</td>
<td>Thread Object Creation System Interface</td>
</tr>
<tr>
<td>6–24</td>
<td>Thread Record Structure</td>
</tr>
<tr>
<td>6–25</td>
<td>Thread Object Deletion System Interfaces</td>
</tr>
<tr>
<td>6–26</td>
<td>Get/Set Thread Information System Interface</td>
</tr>
<tr>
<td>6–27</td>
<td>Signal Thread System Interface</td>
</tr>
<tr>
<td>6–28</td>
<td>Hibernate/Wake Thread System Interface</td>
</tr>
<tr>
<td>6–29</td>
<td>Suspend/Resume Thread System Interface</td>
</tr>
<tr>
<td>6–30</td>
<td>Address Space Creation</td>
</tr>
<tr>
<td>6–31</td>
<td>Initial Thread Entry Point</td>
</tr>
<tr>
<td>6–32</td>
<td>Address Space Initialization</td>
</tr>
<tr>
<td>6–33</td>
<td>Exit Status Object Structure</td>
</tr>
<tr>
<td>6–34</td>
<td>Exit Status Object Creation System Interface</td>
</tr>
<tr>
<td>6–35</td>
<td>Get/Set Exit Status Information System Interface</td>
</tr>
<tr>
<td>6–36</td>
<td>System Thread Creation Executive Interface</td>
</tr>
<tr>
<td>6–37</td>
<td>System Process Creation Executive Interface</td>
</tr>
<tr>
<td>6–38</td>
<td>Raw Quota Interface Definitions</td>
</tr>
<tr>
<td>6–39</td>
<td>Pool Allocator with Quotas Interface</td>
</tr>
</tbody>
</table>

**FIGURES**

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>4–1</td>
<td>Spin Lock Data Structure</td>
</tr>
<tr>
<td>4–2</td>
<td>Thread Dispatcher State Transitions</td>
</tr>
<tr>
<td>4–3</td>
<td>System Time</td>
</tr>
<tr>
<td>5–1</td>
<td>Object ID Format</td>
</tr>
<tr>
<td>5–2</td>
<td>The Big Picture</td>
</tr>
<tr>
<td>5–3</td>
<td>Object Type Descriptor (OTD) Format</td>
</tr>
<tr>
<td>5–4</td>
<td>Object Header Format</td>
</tr>
<tr>
<td>5–5</td>
<td>Object Container Data Structure Relationships</td>
</tr>
<tr>
<td>5–6</td>
<td>Object Container Body Data Structure</td>
</tr>
<tr>
<td>5–7</td>
<td>Container Directory Header Display Index Field Usage</td>
</tr>
<tr>
<td>5–8</td>
<td>Container Object Array Format</td>
</tr>
<tr>
<td>5–9</td>
<td>Address of Object Header</td>
</tr>
<tr>
<td>5–10</td>
<td>Free Object Array Element Format</td>
</tr>
<tr>
<td>5–11</td>
<td>Process Level Container Directory Initialization</td>
</tr>
<tr>
<td>6–1</td>
<td>Complex UJPT Hierarchy</td>
</tr>
</tbody>
</table>

xvii
7-1 Virtual Address Space Layout ................................................. 7-3
7-2 Virtual Address Format ...................................................... 7-4
7-3 Page Table Entry ............................................................. 7-5
7-4 Invalid Page Table Entry .................................................... 7-6
7-5 Valid Page Table Entry ....................................................... 7-7
7-6 Virtual Address to Physical Address Translation ....................... 7-10
7-7 Layout of Virtual Address Space .......................................... 7-12
7-8 Initial Segment 1 Page Table Page ........................................ 7-15
7-9 Memory Management Fault Exception Frame ............................. 7-16
7-10 PFN Database Layout ....................................................... 7-20
7-11 Contents of PFN ORIGINAL_PTE Element ............................. 7-22
7-12 Contents of PFN STATE Element .......................................... 7-22
7-13 Working Set List ............................................................. 7-28
7-14 Contents of Working Set List Entry ...................................... 7-28
7-15 Structures Created by Create Section ................................... 7-32
7-16 Kernel Entry Vector ........................................................ 7-40
7-17 Valid Buffer ................................................................. 7-49
7-18 Page Fault Information ...................................................... 7-50
8-1 Overview of Mica's I/O Architecture .................................... 8-3
8-2 Relationships between Services and Procedures in the I/O Architecture ...................................................... 8-5
8-3 FPU States ........................................................................... 8-11
8-4 I/O Parameter Record Format ................................................. 8-14
8-5 I/O Status Block .................................................................. 8-15
8-6 FP Callbacks ....................................................................... 8-25
8-7 Multiple Callback Tables ....................................................... 8-27
8-8 Accessing an FPU ............................................................... 8-32
8-9 Connection of Channels, FPUs, and FPDs ................................. 8-34
8-10 I/O Request Packet Definition .............................................. 8-36
8-11 I/O Request Packet ............................................................ 8-37
8-12 I/O Request Packet (Passed to Third Function Processor) ......... 8-38
8-13 Binding FPUs Objects ......................................................... 8-54
8-14 Connecting Channels to a Bound FPU .................................... 8-56
8-15 Channel Assigned to Device Unit with DFP ............................. 8-71
8-16 Channel Assigned to Volume with VFP .................................. 8-71
8-17 Channel Assigned to Volume Set with VFP ............................. 8-72

TABLES
5-1 Object Architecture: Terms and Definitions ............................... 5-2
6-1 Get/Set User Information Item Codes ..................................... 6-7
6-2 Get/Set Job Information Item Codes ...................................... 6-13
6-3 Get/Set Process Information Item Codes .................................. 6-20
6-4 Get/Set Thread Information Item Codes ................................... 6-28
6-5 Get/Set Exit Status Information Item Codes ......................... 6-44
7-1 Summary of PFN Fields ....................................................... 7-20
8-1 I/O System Services ........................................................... 8-6
8-2 I/O Executive Services ......................................................... 8-6
8-3 I/O Architecture Procedures ................................................. 8-8
8-4 Function Processor Procedures .............................................. 8-8
8-5 FPU States Codes ............................................................... 8-10
<table>
<thead>
<tr>
<th>Page</th>
<th>Description</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>8-6</td>
<td>Get FPU Information Item Codes</td>
<td>8-16</td>
</tr>
<tr>
<td>8-7</td>
<td>Get Channel Information Item Codes</td>
<td>8-17</td>
</tr>
<tr>
<td>8-8</td>
<td>FPD Object Services and I/O Architecture Procedures</td>
<td>8-41</td>
</tr>
<tr>
<td>8-9</td>
<td>FPU Object Services and I/O Architecture Procedures</td>
<td>8-48</td>
</tr>
<tr>
<td>8-10</td>
<td>Channel Object Services</td>
<td>8-69</td>
</tr>
<tr>
<td>8-11</td>
<td>IRP Services</td>
<td>8-88</td>
</tr>
<tr>
<td>8-12</td>
<td>Synchronizing I/O Services</td>
<td>8-92</td>
</tr>
<tr>
<td>8-13</td>
<td>Cancel I/O Request Services</td>
<td>8-100</td>
</tr>
</tbody>
</table>
CHAPTER 4

THE KERNEL

4.1 Overview

4.1.1 Requirements

The kernel is the lowest layer of software in the system and, as such, is positioned closest to the actual hardware. The kernel is a single layer of code that must implement all interprocessor synchronization, thread dispatching, exception handling, and fork processing. It must also keep the system time and provide services to device drivers for handling interrupts.

The kernel presents a formal interface to the next higher level of software (the executive) that is free of the problems associated with synchronizing various activities on multiple processors and which automatically implements symmetrical multiprocessing (SMP) capabilities.

The kernel attempts to implement no policy. That is the province of the higher levels of software in the system. There are, however, some algorithms that must be implemented in the kernel for efficiency, and therefore, some policy will be included in the kernel. Such a case is the way in which the priority of a thread decays over time. For those cases where it is essential for policy to be located in the kernel, external controls will be provided so that executive software can influence, if not directly control, the actions of the kernel.

4.1.2 Functional Description

4.1.2.1 Environment of the Kernel

The kernel runs in kernel mode, usually at an interrupt priority level (IPL) of 2. This is the priority level at which dispatching occurs. The kernel can be executed simultaneously on all processors in a multiprocessor configuration, and synchronizes access to critical regions as appropriate.

Software within the kernel is not context switchable, whereas all software outside the kernel is always context switchable. In general, executive software is not allowed to raise IPL above 1, or otherwise block context switching, and must use kernel procedures to synchronize its activities.

The kernel is not pageable and cannot take page faults.

All software outside the kernel is written in Pillar. Kernel software is a mixture of Pillar and assembly language. All interfaces to the kernel are defined in Pillar and exported to other programs. It is expected that the size of the kernel will be approximately 8K instructions.
4.1.2.2 Interaction With the Executive

Executive software also runs in kernel mode. It implements system services, memory management, user-level object support, the file system, network access, and device drivers; and it sets system policy.

Executive software communicates with the kernel via a set of data abstractions called kernel objects, and a set of operations that can be performed on these objects. Kernel objects are referred to by address and should not be confused with user objects as defined by the object architecture. Kernel objects are not accessible to user software. An example of a kernel object is an event, which provides a form of synchronization.

There is no firewall protection provided between the kernel and executive software. They both run in kernel mode and can potentially disrupt each other's activity. There is, however, a formal interface between executive software and the kernel, and a well-defined set of rules that must be obeyed.

Normally, the kernel does little or no checking of procedure arguments supplied by the executive; however, debugging software can be conditionally compiled into the kernel to ensure the correctness of calls to kernel procedures. For those cases that the kernel does check argument values for consistency, an error condition is raised via the standard condition mechanism when a parameter value is found to be in error.

4.1.2.3 Primary Kernel Data Structures

The following are the primary data structures defined and used by the kernel:

- The system control block (SCB)—The SCB is an architecturally defined structure that contains an array of exception and interrupt service routine addresses used to service interrupts and exception conditions. The base address of the SCB is stored in the system control block base register (SCBB).

- The processor control block (PB)—The PB contains a collection of processor-specific information. Examples of the information contained in the PB include a pointer to the thread object of the current thread, the processor-specific fork queue header, and counts of the interprocessor interrupts that have occurred. The address of the PB is stored in the processor base register (PRBR), which is defined by the PRISM architecture.

- An array of pointers to the PBs—There is a pointer to the PB for each processor in the system. The index into this array for each processor is stored in the WHAMI register for the processor.

- Spin locks—Spin locks are used to achieve multiprocessor synchronization. In the kernel, spin locks are used to synchronize access to eight kinds of entities:
  1. Dispatcher database
  2. Power-up request queue
  3. Power-up status queue
  4. VAX port queues
  5. Device work queues
  6. Active I/O interrupts
  7. Processor request
  8. Kernel debugger

- Kernel objects—Kernel objects are data abstractions that are necessary to control processor execution and synchronization. Kernel objects parallel objects as defined in Chapter 5, Object Architecture, but are not directly available to user software and are addressed by pointers rather than object IDs. Kernel objects are divided into two categories, dispatcher objects and control objects. The kernel objects are:
<table>
<thead>
<tr>
<th>Dispatcher objects</th>
<th>Control objects</th>
</tr>
</thead>
<tbody>
<tr>
<td>Event</td>
<td>AST</td>
</tr>
<tr>
<td>Mutex</td>
<td>Device work queue</td>
</tr>
<tr>
<td>Queue</td>
<td>Interrupt</td>
</tr>
<tr>
<td>Semaphore</td>
<td>Power-up request</td>
</tr>
<tr>
<td>Timer</td>
<td>Power-up status</td>
</tr>
<tr>
<td>Thread</td>
<td>Process</td>
</tr>
<tr>
<td></td>
<td>VAX port queue</td>
</tr>
</tbody>
</table>

- Dispatcher database—The dispatcher database is used when choosing which threads should be active at any point in time. The database is a collection of data structures that contains information such as a list of threads ready for execution, and a record of which processors are executing threads at which priority levels.

- Timer queue—The timer queue is a binary tree of timer objects that are each set to expire at a specified time.

- Power-up request and status queues—The power-up request and status queues are used to notify threads when a power recovery interrupt is received by PRISM hardware.

- Performance data—The kernel collects and stores performance data in various private data structures.

### 4.1.2.4 Primary Kernel Functions

The primary functions of the kernel include:

- Multiprocessor coordination—To coordinate the activity of multiple processors the kernel uses spin locks for synchronization and interprocessor interrupts for notifying other processors of work to be done. Executive code outside the kernel can use either spin locks or mutex objects to implement mutual exclusion.

- Thread dispatching—The kernel supports 64 levels of thread priority. The highest 16 levels are referred to as real-time priorities and the lowest 48 levels as class priorities. The kernel implements dispatching, which chooses exactly which thread to execute next. Scheduling, which selects the threads that are eligible for execution, is the province of higher levels of software.

- AST Processing—The kernel provides services for queuing and delivering asynchronous system traps (ASTs) to target threads. A combination of software state and hardware registers is used to determine the correct time to interrupt thread execution.

- Interval timer support and the system time—The interval timer is used by the kernel for maintaining the system time, accumulating accounting and performance information, updating thread quantum, and timer queue maintenance. The system time is maintained as a quadword count of 100ns intervals and is initialized to zero when the system is booted.

- Address space number (ASN) Management—The kernel provides for complete management of the assignment of address space numbers (ASNs). ASNs are used to tag translation buffer entries and therefore avoid flushing at every context switch.

- Powerfail Recovery—Powerfail recovery support is provided by the kernel via power-up request and status objects. In conjunction with raising IPL, these objects provide a driver thread with the capability to interrupt its execution and/or have a status variable set when a power recovery interrupt is received by PRISM hardware. Power-up status objects may only be used directly by kernel-mode code. Power-up request objects are intended primarily for use by driver threads, but can also be provided to user-mode programs via executive objects.
4.1.2.5 Performance Data Collection

The kernel collects various categories of performance data during its execution so that both the designers and users of the system can analyze and improve its performance. The data structures required to record this data are private to the kernel and, therefore, are not directly accessible to executive software. Executive software can retrieve the following data, however, by calling a kernel procedure that returns the desired category of data:

- Number of currently computable and waiting threads
- Processor fork queue depth
- Context switch headway
- Number of interprocessor interrupts (for each kind of request)
- Interrupt data for an interrupt vector
- Contention data for device work queues and mutexes
- Processor mode data
- Dispatcher object wait queue depth

4.2 System Control Block

The system control block (SCB) is an architecturally defined structure whose base address is held in the system control block base (SCBB) register. The SCB contains an array of exception and interrupt service routine addresses. When an interrupt or exception condition arises, the processor reads the appropriate address from the SCB, saves the current processor status and program counter on the kernel stack, and vectors execution to the specified location. There is one SCB for the entire system.

4.3 Processor Control Block

The processor base register (PRBR) is an architecturally defined register that is used by the kernel to locate processor-specific information. When the system is initialized, a processor control block (PB) is allocated for each processor. The address of this control block is then stored in PRBR. Thereafter, the kernel can obtain the address of the current processor's PB by executing an MFPR instruction specifying the register PRBR.

The following information is stored in the processor control block (PB):

- A pointer to the thread object of the current thread that is executing on the processor (pb$now_thread).
- A pointer to the thread object of the next thread that has been selected for execution on the processor (pb$next_thread).
- A pointer to the thread object of the last thread to use the vector registers on the respective processor (pb$last_thread).
- A pointer to the thread object of the idle thread for the respective processor (pb$idle_thread).
- The address space sequence number and the number of address space numbers (ASNs) supported by the respective processor (pb$asn_sequence and pb$asn_number).
- The processor-specific fork queue header and performance data (pb$fork_first, pb$fork_count, pb$fork_depth).
- The processor-specific request spin lock, request summary, and related information (pb$request_lock, pb$request_summary, pb$request_value).
- The processor-specific power recovery fork block that is used to initiate processing of the power recovery request list at fork level (pb$power_fork).
• The processor-specific quantum end fork block that is used to initiate quantum end processing at fork level (pb$quantum_fork).

• The processor-specific timer queue fork block that is used to invoke examination of the timer queue at fork level (pb$timer_fork).

• A count of the number of interval timer interrupts that have occurred while the processor mode was kernel (pb$kernel_ticks).

• A count of the number of interval timer interrupts that have occurred while the processor mode was user. (pb$user_ticks).

• A count of the number of interprocessor interrupts that have occurred for each of the request types (pb$asts, pb$translation_flushes, pb$dispatches, pb$freezes, pb$instruction_flushes, and pb$translation_flush_ones).

4.4 Information Addressed by Processor Index

The kernel maintains an array of pointers that are addressed by processor index. A processor's index is a number from 0 to n-1, where n is the number of processors in a system. The index of the current processor can be obtained by executing an MFPR instruction that specifies the internal processor register WHAMI (Who-Am-I).

This array contains the addresses of the processor control blocks (PBs) for each of the processors in a system and allows software running on one processor to address the processor-specific information of another processor.

The extent of this array is 32, which is the maximum number of processors supported by the kernel.

4.5 Kernel Use of Hardware Interrupt Priority Levels

The kernel uses the hardware interrupt priority levels (IPLs) as follows:

<table>
<thead>
<tr>
<th>IPL</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Thread execution</td>
</tr>
<tr>
<td>1</td>
<td>AST delivery interrupt</td>
</tr>
<tr>
<td>2</td>
<td>Dispatch interrupt and fork processing</td>
</tr>
<tr>
<td>3</td>
<td>Wake system debugger</td>
</tr>
<tr>
<td>4</td>
<td>Device I/O interrupt service routines</td>
</tr>
<tr>
<td>5</td>
<td>Device I/O interrupt service routines</td>
</tr>
<tr>
<td>6</td>
<td>Interprocessor interrupts, interval clock interrupt</td>
</tr>
<tr>
<td>7</td>
<td>Powerfail recovery interrupt, machine check interrupt</td>
</tr>
</tbody>
</table>

4.6 Multiprocessor Synchronization

During various stages of its execution, the kernel must ensure that only one processor is allowed to be active in a given sequence of code (for example, dispatching a thread or setting an event). During these periods, the kernel must prevent the execution of code on other processors that could potentially be simultaneously accessing or modifying the same data. Spin locks are the mechanism by which this is achieved.

Spin locks are used when mutual exclusion must exist across all processors and context switching cannot take place. A spin lock takes its name from the fact that, while waiting on a spin lock, software continually tries to gain entry to a critical region and makes no progress until it succeeds. This is accomplished by implementing a test and set operation on an interlock variable using the RMALI instruction. When software executes a test and set operation and finds the previous state of the lock free, it is granted entry to a critical region. If, however, software finds the previous state of the lock set, it just repeats the test until it finds the previous state free.
There are eight kinds of spin locks used within the kernel:

1. **Dispatcher database**
   The dispatcher database describes the scheduling state of the system. Whenever a change in system state is possible or the current system state is examined, the dispatcher database spin lock must be acquired. State changes can occur as the result of satisfying a thread Wait condition, insertion of an entry in a queue, setting an event, expiration of a timer queue entry, and so on.

2. **Power-up request queue**
   The power-up request queue enables a thread to be notified when a power failure has occurred. The power-up request spin lock must be acquired when an entry is inserted in the queue and while processing the queue after a power-up interrupt has occurred.

3. **Power-up status queue**
   The power-up status queue provides the capability to have a specified boolean variable set to a true value when a power-up interrupt is received. The power-up status spin lock must be acquired when an entry is inserted in the queue and while processing the queue after a power-up interrupt has occurred.

4. **VAX port queues**
   Each VAX port device that is directly connected to a PRISM system has one or more VAX-type interlocked queues. The secondary Interlock bit is used on the PRISM processor side to synchronize updates. Whenever an entry is being inserted or removed from the queue, the corresponding VAX queue spin lock must be acquired. No spin lock count is maintained for these queues (see below).

5. **Device work queues**
   A device work queue is used to pass information between an interrupt service routine and a device driver. Normally, interrupt service routines insert an entry in a device work queue and then set an event. Driver threads wait on an event and remove entries from the device work queue. Whenever an entry is being inserted or removed from the queue, the corresponding work queue spin lock must be acquired.

6. **I/O interrupt synchronization**
   Each connected I/O interrupt has a spin lock that is used to provide mutual exclusion between the execution of the interrupt service routine and critical regions of device driver code (for example, the loading of device registers). When a device interrupt occurs the corresponding spin lock is acquired. Driver code must also acquire the spin lock when synchronization with the interrupt service routine is required.

7. **Processor request**
   Each processor has a request summary which is used by other processors to signal an action to be performed. Typically, kernel code running on one processor acquires another processor's request summary spin lock, updates information in memory, sets appropriate request bits in the request summary, sends an interprocessor interrupt to the target processor, and then releases the processor's request summary spin lock.

8. **Kernel debugger**
   The kernel debugger is used to debug the kernel which can be in execution on several processors simultaneously. Whenever the debugger is entered it must acquire the debugger spin lock to ensure that it is not reentered on another processor.
Spin locks can only be operated on from a safe interrupt priority level. This means that an attempt to acquire a particular spin lock cannot be made unless the processor is running at the highest IPL from which any other acquire request for the same spin lock could be made on the same processor. For instance, if a spin lock was successfully acquired at IPL 2, and then another attempt was made to acquire the same spin lock at IPL 3 on the same processor, a deadlock situation would arise, because the code running at IPL 2 would never get a chance to run and release the spin lock.

The highest IPL that the dispatcher database is acquired from is IPL 2. Therefore, when attempting to acquire the dispatcher database spin lock, the processor IPL must be 2 and must remain at 2 until the spin lock is released.

The power-up request queue is processed as a result of the power recovery fork action routine which is executed at IPL 2. While the power recovery queue is being processed the dispatcher database spin lock must be repeatedly acquired and released. Therefore, when attempting to acquire the power-up request queue spin lock, the processor IPL must be at 2 and must remain at 2 until the spin lock is released.

The power-up recovery interrupt occurs at IPL 7. Therefore, when attempting to acquire the power-up status queue spin lock, the processor IPL must be 7 and remain at 7 until the spin lock is released.

VAX port queues are accessed from kernel code, from device interrupt service routines running at IPL 4 or 5, and from VAX port controllers. Even though the highest IPL from which a PRISM processor accesses one of these queues is 5, the IPL must be 7 and remain at 7, while an entry is being inserted or removed from the respective queues. This ensures that VAX port controller code is not prevented from manipulating the queue while the PRISM processor is possibly servicing other unrelated interrupts. It also prevents a power failure sequence during queue operations that could possibly leave the queue in a locked state.

Device work queues are accessed from kernel code and from device interrupt service routines running at IPL 4 or 5. Therefore, when attempting to acquire a work queue spin lock, the IPL must be raised to the highest level from which the queue is accessed and remain at that level until the corresponding spin lock is released.

I/O interrupt service routines are executed in response to interrupts that occur at their respective IPL. Device drivers need to interlock with the execution of their respective interrupt service routines to manipulate device-specific data structures. These interrupts occur at levels 4 and 5. Therefore, when attempting to synchronize the execution of an interrupt service routine with device driver code, the IPL must be raised to that of the interrupting source and remain at that level until the spin lock is released.

Processor requests are examined by the target processor in the interprocessor interrupt service routine, which runs at IPL 6. However, in order to debug code running at all IPL levels, the IPL must be raised to 7 when attempting to acquire the processor request spin lock, and must remain at 7 until the spin lock is released.

When the debugger is entered it must raise the IPL to 7 to ensure that no other actions take place before debugging of the kernel actually begins. Therefore, when attempting to acquire the debugger spin lock, the IPL must be at 7 and must remain at 7 until the debugger spin lock is released.

A spin lock is longword aligned. The format of a spin lock is shown in Figure 4-1.

**Figure 4-1: Spin Lock Data Structure**

![Spin Lock Data Structure](image)
All lock and unlock operations on a spin lock must be performed with a RMALI instruction. The low-order bit (LCK) of the spin lock is the Lock bit. If the bit is clear the lock is free; if it is set the lock is owned. A 31-bit count is included in the spin lock to count the number of times a lock attempt has failed. This data will be useful in future refinements of the kernel and will give us a measure of lock conflict.

The instruction sequence to lock a spin lock is:

```
SUB    #2, R0, R5 ; set mask value
OR     #1, R0, R6 ; set addend value
10$:
LDA    spin_lock, R4 ; get address of spin lock
RMALI
BLBC   R4, 30$ ; if lbc lock was free
ADD    #3, R0, R6 ; set initial count value
20$:
LDL    spin_lock, R4 ; read spin lock value
ADD    #2, R5, R6 ; update spin count
BLBC   R4, 10$ ; if lbc lock is free
BEQ    R0, 20$ ; loop until lock is free
30$:
.
.
; start of critical region
.
.
```

This instruction sequence sets the spin lock. If the lock was previously free, then the critical region is entered. Otherwise, the spin count is initialized, and a loop consisting solely of noninterlocked instructions is executed until the lock is free. The interlocked sequence is then repeated, and the accumulated spin count is added to the lock spin count.

The instruction sequence to unlock a spin lock is:

```
SUB    #2, R0, R5 ; set mask value
OR     R0, R0, R6 ; set addend value
LDA    spin_lock, R4 ; get address of spin lock
RMALI
```

### 4.6.1 Executive Multiprocessor Synchronization

Executive software outside the kernel also has the requirement to synchronize access to resources in a multiprocessor environment. Unlike the kernel, however, executive software can use mutex objects (see Section 4.8.1.2), as well as spin locks, to implement mutually exclusive access. Mutex objects allow the processor to be redispached and should be used when the wait time or access time to a resource is likely to be lengthy (e.g., greater than 200 microseconds). Spin locks should be used when the wait time is small and access to the resource will not extend over a large interval of time.

Executive use of spin locks could cause serious maintenance and reliability problems if not controlled. In particular, no deadlock detection is performed and dispatching is disabled while the executive owns a spin lock. Therefore certain rules must be followed by executive software when using spin locks. The rules are as follows:

- The code within a critical region guarded by an executive spin lock must not be pageable and must not make any references to pageable data.
- An executive spin lock can only be acquired from IPL 0, IPL 1 or IPL 2.
- The code within a critical region guarded by an executive spin lock cannot call any external procedures, nor can it generate any conditions or exceptions.
- The code within a critical region guarded by an executive spin lock cannot raise or lower IPL.

An executive spin lock can be acquired by calling a kernel function that has the following declaration:

```
PROCEDURE k$acquire_lock (  
    BIND Lock : k$lock;  
) RETURNS k$sip1;
```

4-8 The Kernel
Parameters:

lock The executive spin lock to be acquired.

The previous IPL is saved, the IPL is raised to dispatcher level, and the specified spin lock is acquired. The previous IPL is returned as the function value and must be specified as an argument when the spin lock is released.

After acquiring an executive spin lock, executive software executes at IPL 2 with dispatching disabled.

An executive spin lock can be released by calling a kernel procedure that has the following declaration:

```pro
PROCEDURE
  k$release_lock (  
    BIND Lock : k$lock;
    IN ipl : k$ipl;
    IN wait : boolean = false;
  );
```

Parameters:

lock The executive spin lock to be released.

ipl The IPL at which the executive spin lock was acquired.

wait A boolean variable that specifies whether the call to k$release_lock will be immediately followed by a call to one of the Wait functions.

If the value of the wait parameter is FALSE, then the specified spin lock is released and the IPL is set to the level specified by the ipl parameter.

If the value of the wait parameter is TRUE, then the specified spin lock is released, the dispatcher database spin lock is acquired, the previous IPL is stored in the current thread object and a return to the caller is executed at IPL 2 with the dispatcher database locked. For this case, the call to k$release_lock MUST be IMMEDIATELY followed by a call to one of the Wait functions. This capability can be used to ensure that a context switch does not occur between the releasing of the spin lock and the subsequent wait operation.

### 4.7 System-Wide Data Structures

The kernel maintains several data structures that are system-wide. These structures contain system-wide performance information and data that is essential to the operation of the system.

The following information is maintained on a system-wide basis:

- The system control block (SCB)
- Dispatcher database
- Context switch "headway" performance data
- Wait state performance data
- The power-up request object queue list head
- The power-up status object queue list head
- The timer queue list head
4.8 Kernel Objects

The kernel provides support for the data abstractions that are necessary to control processor execution and synchronization. These data abstractions are referred to as *kernel objects*. They parallel objects as defined in the system object architecture, but are not directly available to user software and are addressed by pointers rather than object IDs.

Kernel objects are created by first allocating an appropriate data structure using a hidden type and then calling a kernel procedure to initialize the particular object type structure. Subsequent manipulation of the object is accomplished via object-specific procedures provided by the kernel.

Kernel objects that are used to synchronize thread execution and which can be specified in a kernel Wait function are referred to as dispatcher objects. Other kernel objects not included in this class are referred to as control objects.

The following sections describe each kernel object, its initialization, and a set of object-specific operations that can be performed on each object type.

4.8.1 Dispatcher Objects

A *dispatcher object* is a kernel object that is used to control and synchronize thread access to data structures and external events. Each dispatcher object has a state which is either Signaled or Not-Signaled. When a dispatcher object is in a Signaled state, any Wait operation that specifies the object will be satisfied and the invoking thread will not wait. A dispatcher object that is in a Not-Signaled state, however, causes the execution of any thread that specifies the object in a Wait operation to be suspended until the object attains a Signaled state.

The following dispatcher objects are supported by the kernel:

<table>
<thead>
<tr>
<th>Dispatcher Object</th>
<th>Type Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Event</td>
<td>k$dispatcher_object(k$c_event)</td>
</tr>
<tr>
<td>Mutex</td>
<td>k$dispatcher_object(k$c_mutex)</td>
</tr>
<tr>
<td>Queue</td>
<td>k$dispatcher_object(k$c_queue)</td>
</tr>
<tr>
<td>Semaphore</td>
<td>k$dispatcher_object(k$c_semaphore)</td>
</tr>
<tr>
<td>Timer</td>
<td>k$dispatcher_object(k$c_timer)</td>
</tr>
<tr>
<td>Thread</td>
<td>k$dispatcher_object(k$c_thread)</td>
</tr>
</tbody>
</table>

Note that dispatcher objects cannot be manipulated from interrupt procedures, with the exception of calling the *k$set_event_from_interrupt* procedure.

4.8.1.1 Event Object

An *event object* is used to record and synchronize the occurrence of an event with some action that is to be performed.

There are two kinds of event objects:
1. Notification
2. Synchronization

Notification event objects are set (Signaled state) upon the occurrence of some related event and remain set until explicitly cleared. Synchronization event objects are set (Signaled state) upon the occurrence of a related event, but are automatically cleared when a Wait is satisfied. Both kinds of event objects can also be cleared explicitly.
A notification event is typically used where it is desirable for a number of threads to wait on a single event and all proceed when the event is set (for example, an implicit resource Wait such as the allocation of nonpaged pool).

A synchronization event is used when it is desirable to be able to set the event multiple times but only satisfy the Wait for a single thread (for example, a wake/hibernate, suspend/resume, or fast mutex mechanism).

An event object is initialized by calling a kernel procedure that has the following declaration:

```plaintext
PROCEDURE
  k$initialize_event (
    IN event : POINTER k$dispatcher_object; !k$c_event);
    IN kind : k$event_type;
    IN state : boolean;
  );
```

Parameters:
- `event` The virtual address of a dispatcher object of type event.
- `kind` The kind of event to initialize: `k$c_synchronization` or `k$c_notification`.
- `state` The initial state of the event.

Waiting on an event object of the notification type, waits until the event is set (Signaled state). Satisfying the Wait for an event object of this type does not change the state of the event object.

Waiting on an event object of the synchronization type, also waits until the event is set (Signaled state). However, satisfying the Wait causes the event object to be cleared (Not-Signaled state).

An event object can be set to a Signaled state by calling a kernel function that has the following declaration:

```plaintext
PROCEDURE
  k$set_event ( 
    IN event : POINTER k$dispatcher_object; !k$c_event);
    IN increment : k$priority_increment;
    IN wait : boolean = false;
  ) RETURNS boolean;
```

Parameters:
- `event` The virtual address of a dispatcher object of type event.
- `increment` The priority increment that is to be applied if setting the event causes a Wait to be satisfied.
- `wait` A boolean value that specifies whether the call to the `k$set_event` function will be immediately followed by a call to one of the Wait functions.

The previous state of the event object is returned as the function value. If the value of the `wait` parameter is `TRUE`, then the call to the `k$set_event` function MUST be IMMEDIATELY followed by a call to one of the Wait functions, since a return to the caller is executed at IPL 2 with the dispatcher database locked. This capability can be used to ensure that a context switch does not occur between the setting of an event and the subsequent Wait operation.

An event object can be pulsed to a Signaled state and then cleared by calling a kernel function with the following declaration:

```plaintext
PROCEDURE
  k$pulse_event ( 
    IN event : POINTER k$dispatcher_object; !k$c_event);
    IN increment : k$priority_increment;
    IN wait : boolean = false;
  ) RETURNS boolean;
```
Parameters:

- **event**: The virtual address of a dispatcher object of type event.
- **increment**: The priority increment that is to be applied if setting the event causes a Wait to be satisfied.
- **wait**: A boolean value that specifies whether the call to the \$pulse_event function will be immediately followed by a call to one of the Wait functions.

The previous state of the event object is returned as the function value. This function sets the event, tries to satisfy a Wait for the event, and then clears the event.

The Signal state of an event object can be cleared (set to a Not-Signaled state) or read by calling one of the following kernel functions:

```c
PROCEDURE
  k$clear_event (  
    IN event : POINTER k$dispatcher_object; !(k$c_event);  
  ) RETURNS boolean;

PROCEDURE
  k$read_event (  
    IN event : POINTER k$dispatcher_object; !(k$c_event);  
  ) RETURNS boolean;
```

Parameters:

- **event**: The virtual address of a dispatcher object of type event.

The previous Signal state (current Signal state in the case of k$read_event) of the event is returned as the function value.

### 4.8.1.2 Mutex Object

A **mutex object** is used to control exclusive access to a resource. The state of a mutex is controlled by a count. When the count is 1 (Signaled state), the mutex is not owned, and access to the resource associated with the mutex can be allowed. Granting ownership of a mutex causes the count to be decremented to 0 (Not-Signaled state), whereupon no other thread is able to gain access to the resource until the owner thread releases the mutex.

A mutex object is initialized by calling a kernel procedure that has the following declaration:

```c
PROCEDURE
  k$initialize_mutex (  
    IN mutex : POINTER k$dispatcher_object; !(k$c_mutex);  
    IN level : integer;  
  );
```

Parameters:

- **mutex**: The virtual address of a dispatcher object of type mutex.
- **level**: The level number that is to be assigned to the mutex object.

Mutex objects are initialized with an initial count of 1 (Signaled state).

Waiting on (acquiring) a mutex suspends the execution of the invoking thread until the mutex count is 1. Satisfying the Wait causes the count to be decremented to 0 and grants ownership of the mutex to a waiting thread. When ownership of a mutex is granted, normal ASTs for kernel mode are automatically disabled. If the granted mutex is the first one owned by the invoking thread, then the thread's current priority is saved and then raised to the maximum of its current priority and the lowest real-time priority. This ensures that the thread will have a high execution priority while it executes code in a critical section. Granting ownership of a mutex also prevents the thread's process from leaving the balance set by incrementing the count of mutexes owned by the process; see Section 4.8.1.7.2.2.

4-12 The Kernel
Level numbers are used to guarantee deadlock-free operation. A list of the mutexes that have been acquired (waited on) and their highest level number is maintained in the thread object. If a thread attempts to acquire a mutex with a level number less than or equal to the highest level number of a mutex already owned by the thread, an error condition is raised.

It is intended that the design of the system will be deadlock free. But history has shown that as future maintenance occurs and people dabble in places they have little understanding, things break. Level checking is intended to be a low overhead way to automatically catch this kind of error. 

Threads are allowed to acquire a mutex that they already own multiple times. This allows for recursive code that acquires and releases a mutex. Before such a mutex is considered unowned, however, it must be released the same number of times that it was acquired. Level number checks are not performed for mutexes that are recursively obtained (that is, already owned by the thread when the attempt is made to acquire the mutex).

Mutexes can only be acquired and released by kernel-mode code. Furthermore, executive software cannot export mutexes to user mode, nor can any thread acquire a mutex in kernel mode and then return to user mode.

A mutex can be released by calling a kernel procedure that has the following declaration:

```
PROCEDURE
  k$release_mutex
  
  IN mutex : POINTER k$dispatcher_object; !(k$c_mutex);
  IN increment : k$priority_increment;
  IN wait : boolean = false;
  ) RETURNS integer;
```

Parameters:

- **mutex**: The virtual address of a dispatcher object of type mutex.
- **increment**: The priority increment that is to be applied if releasing the mutex causes a Wait to be satisfied.
- **wait**: A boolean value that specifies whether the call to the \( k$release_mutex \) function will be immediately followed by a call to one of the Wait functions.

The previous state of the mutex (as indicated by the count) is returned as the function value. If the value of the wait parameter is TRUE, then the call to the \( k$release_mutex \) function MUST be IMMEDIATELY followed by a call to one of the Wait functions, since a return to the caller is executed at IPL 2 with the dispatcher database locked. This capability can be used to ensure that a context switch does not occur between the release of a mutex and the subsequent Wait operation.

Releasing a mutex causes the mutex count to be incremented. If the resultant count is less than 1 (that is, the mutex was recursively acquired), then the mutex remains in the Not-Signaled state. Otherwise, the state of the mutex becomes Signaled and it is removed from the releasing thread's mutex list. If the mutex is the last one in the list, then the previous thread priority is restored; and if the thread's kernel-mode AST queue is not empty, a kernel-mode AST is requested by writing the ASTRR register.

Removal of a mutex from a thread's mutex list also causes the count of mutexes owned by the thread's process to be decremented. If the resultant count is 0 and the thread's process is inactive, then the process activity event specified by the current thread's process object is set (this event is used by the balance set scheduler to synchronize removal of a process from the balance set), and the current processor is rescheduled.

If an attempt is made to release a mutex that is not owned by the thread executing the release operation, then an error condition is raised.

The count of a mutex (Signal state) can be read by calling a kernel function that has the following declaration:

```
PROCEDURE
  k$read_mutex
  
  IN mutex : POINTER k$dispatcher_object; !(k$c_mutex);
  ) RETURNS integer;
```

The Kernel 4-13
Parameters:

mutex

The virtual address of a dispatcher object of type mutex.

The mutex count (Signaled state) is returned as the function value.

### 4.8.1.2.1 Mutex Performance Data

Two counters are maintained for each mutex. One counter signifies the total number of times the mutex has been acquired, and the other signifies the number of times that an attempt to acquire the mutex resulted in suspension of the execution of the acquiring thread. These counters can be read via a performance data collection procedure (see Section 4.16 for a description of performance data collection).

### 4.8.1.3 Queue Object

A queue object is a repository for queue entries and is used to synchronize activity between producer and consumer threads. The state of a queue object is controlled by the number of entries in the queue. If the queue is empty, the queue object is in the Not-Signaled state. Otherwise, the queue object is in the Signaled state (queue contains at least one entry).

A queue object is initialized by calling a kernel procedure that has the following declaration:

```plaintext
PROCEDURE k$initialize_queue (  
   IN queue : POINTER k$dispatcher_object; !(k$c_queue);  
);
```

Parameters:

queue

The virtual address of a dispatcher object of type queue.

Waiting on a queue object suspends the execution of the waiting thread until the queue contains at least one entry. Satisfying the Wait for a queue object does not change the state of the queue object.

An entry can be inserted at the head or tail of a queue object by calling one of the following kernel procedures:

```plaintext
PROCEDURE k$insert_head_queue (  
   IN queue : POINTER k$dispatcher_object; !(k$c_queue);  
   IN increment : k$priority_increment;  
   IN entry : POINTER k$list_entry;  
   IN wait : boolean = false;  
 ) RETURNS boolean;

PROCEDURE k$insert_tail_queue (  
   IN queue : POINTER k$dispatcher_object; !(k$c_queue);  
   IN increment : k$priority_increment;  
   IN entry : POINTER k$list_entry;  
   IN wait : boolean = false;  
 ) RETURNS boolean;
```

Parameters:

queue

The virtual address of a dispatcher object of type queue.

increment

The priority increment that is to be applied if inserting in the queue causes a Wait condition to be satisfied.

entry

The virtual address of the queue entry that is to be inserted at the head or tail of the queue.

wait

A boolean value that specifies whether the call to the k$insert_head_queue or k$insert_tail_queue function will be immediately followed by a call to one of the Wait functions.
If the previous state of the queue object was Not-Signaled, then TRUE is returned as the function value; otherwise, FALSE is returned. If the value of the wait parameter is TRUE, then the call to the \texttt{k$\text{insert\_head\_queue}} or \texttt{k$\text{insert\_tail\_queue}} function MUST be IMMEDIATELY followed by a call to one of the Wait functions, since a return to the caller is executed at IPL 2 with the dispatcher database locked. This capability can be used to ensure that a context switch does not occur between a queue insertion and the subsequent Wait operation.

An entry can be removed from the head or tail of a queue object by calling one of the following kernel functions:

```plaintext
PROCEDURE
   \texttt{k$\text{remove\_head\_queue}} ( 
      \texttt{IN queue : POINTER k$\text{dispatcher\_object}; !k$c\_queue); }
   \texttt{) RETURNS POINTER k$\text{list\_entry};}

\texttt{k$\text{remove\_tail\_queue}} ( 
   \texttt{IN queue : POINTER k$\text{dispatcher\_object}; !k$c\_queue); }
   \texttt{) RETURNS POINTER k$\text{list\_entry};}
```

Parameters:

\texttt{queue} \quad The virtual address of a dispatcher object of type \texttt{queue}.

If the queue is empty, a null pointer is returned as the function value. Otherwise, a pointer to the entry that is removed from the queue is returned. If the last entry is removed from the queue, the state of the queue object becomes Not-Signaled.

The current number of entries in a queue object (Signal state) can be read by calling a kernel function that has the following declaration:

```plaintext
PROCEDURE
   \texttt{k$\text{read\_queue}} ( 
      \texttt{IN queue : POINTER k$\text{dispatcher\_object}; !k$c\_queue); }
      \texttt{) RETURNS integer;}
```

Parameters:

\texttt{queue} \quad The virtual address of a dispatcher object of type \texttt{queue}.

The number of entries currently in the queue (Signal state) is returned as the function value.

### 4.8.1.4 Semaphore Object

A \textit{semaphore object} is used to control access to a resource, but not necessarily in a mutually exclusive fashion. A semaphore acts as a gate through which a variable number of threads can pass concurrently, up to a specified limit. The gate is open (Signaled state) as long as there are resources available. When the number of resources that may be concurrently in use has been exhausted, the gate is closed (Not-Signaled state). The gating mechanism of a semaphore is implemented by a counter. Waiting on a semaphore waits until a resource is available and decrements the count. Releasing a semaphore increments the count and allows another thread to pass through the gate.

A semaphore object is initialized by calling a kernel procedure that has the following declaration:

```plaintext
PROCEDURE
   \texttt{k$\text{initialize\_semaphore}} ( 
      \texttt{IN semaphore : POINTER k$\text{dispatcher\_object}; !k$c\_semaphore); }
      \texttt{IN count : integer; }
      \texttt{IN maximum : integer; }
      \texttt{)};
```
Parameters:

semaphore The virtual address of a dispatcher object of type semaphore.
count The initial count of the semaphore.
maximum The maximum count the semaphore can attain.

If the value of the count parameter is greater than the value of the maximum parameter, an error condition is raised.

Waiting on (acquiring) a semaphore suspends the execution of the invoking thread until the semaphore count is greater than zero. Satisfying the Wait causes the count to be decremented.

\ A semaphore with an initial count of 0 or 1 and a maximum count of 1 does not behave exactly the same as a mutex. Special operations are performed for mutexes, such as automatically disabling ASTs, deadlock detection via level numbers, raising thread priority, and locking the thread’s process in the balance set. None of these capabilities are provided with semaphores. \n
A semaphore is released by calling a kernel procedure that has the following declaration:

PROCEDURE
k$release_semaphore (  
    IN semaphore : POINTER k$dispatcher_object; !(k$c_semaphore);  
    IN increment : k$priority_increment;  
    IN wait : boolean = false;  
) RETURNS integer;

Parameters:

semaphore The virtual address of a dispatcher object of type semaphore.
increment The priority increment that is to be applied if releasing the semaphore causes a Wait condition to be satisfied.
wait A boolean value that specifies whether the call to the k$release_semaphore function will be immediately followed by a call to one of the Wait functions.

Releasing a semaphore causes the number of available resources to be incremented. If an attempt is made to release a semaphore beyond the maximum number of resources that were specified when the semaphore was initialized, an error condition is raised.

The previous state of the semaphore is returned as the function value. If the value of the wait parameter is TRUE, then the call to the k$release_semaphore function MUST be IMMEDIATELY followed by a call to one of the Wait functions, since a return to the caller is executed at IPL 2 with the dispatcher database locked. This capability can be used to ensure that a context switch does not occur between releasing a semaphore the subsequent Wait operation.

The current count of a semaphore (Signal state) can be read by calling a kernel function that has the following declaration:

PROCEDURE
k$read_semaphore (  
    IN semaphore : POINTER k$dispatcher_object; !(k$c_semaphore);  
) RETURNS integer;

Parameters:

semaphore The virtual address of a dispatcher object of type semaphore.

The current count of the semaphore (Signal state) is returned as the function value.
4.8.1.5 Timer Object

A timer object is used to record the passage of time. Timers are set to a specified time and then expire when the time comes due. When the timer is set, the kernel inserts the timer object in the timer queue and sets its state to Not-Signaled. At expiration of the timer, the kernel removes the timer object from the timer queue, sets its state to Signaled, and optionally queues an AST to the thread that set the timer (see Section 4.10 for a description of AST processing).

A timer object is initialized by calling a kernel procedure that has the following declaration:

```
PROCEDURE
  k$initialize_timer (  
    IN timer : POINTER k$dispatcher_object; !(k$c_timer);  
  );
```

Parameters:

- **timer**: The virtual address of a dispatcher object of type timer.

The initial state of the timer is set to Not-Signaled.

Waiting on a timer suspends the execution of the invoking thread until the timer has expired. Satisfying the Wait for a timer object does not change the state of the timer object.

A thread can use timers to execute specific actions at fixed points in time or at various intervals. To accomplish this, the thread creates a timer object, sets the timer to expire at a specified time, then waits on the timer object and/or specifies an AST object which is to be queued when the time has expired. At the expiration of the timer, the state of the timer object is set to Signaled, and the thread continues execution.

A timer object can be set to expire at a specified time by calling a kernel procedure that has the following declaration:

```
PROCEDURE
  k$set_timer (  
    IN timer : POINTER k$dispatcher_object; !(k$c_timer);  
    IN due_time : large_integer;  
    IN ast : POINTER k$control_object OPTIONAL; !(k$c_ast);  
  );
```

Parameters:

- **timer**: The virtual address of a dispatcher object of type timer.
- **due_time**: The time at which the timer is to expire.
- **ast**: An optional parameter which specifies the virtual address of a control object of type AST.

The expiration time is computed, the state of the timer is set to Not-Signaled, and the timer object is inserted in the timer queue. If the timer object was already in the timer queue, then it is removed by performing an implicit cancellation, and reinserted in the queue.

Due time is expressed in system time units which are 100ns intervals. Absolute time is expressed as a positive time, whereas a delta time is specified as a negative time. When a delta time is specified, the actual absolute time is calculated by first negating the delta time, then adding it to the current system time. If an absolute time is specified that is prior to the current system time, then the timer is still inserted in the timer queue and expires at the first interval timer interrupt.

If an AST object is specified, then an AST is queued when the timer expires. The AST is delivered in accordance with the parameters that were specified when the AST object was initialized (see Section 4.8.2.1 for a description of AST objects).

A activity bit is maintained by the kernel in the AST object to prevent multiple insertion of the object in an AST queue. This bit is set when the AST object is inserted in an AST queue and cleared when the AST is delivered.
Repeatable timers can be implemented by specifying a kernel routine that is to be called just prior to actually delivering the specified AST; see Section 4.8.2.1.

A timer that was previously set can be canceled. Canceling a timer removes it from the timer queue. A timer object can be canceled by calling a kernel procedure that has the following declaration:

```
PROCEDURE k$cancel_timer (
    IN timer : POINTER k$dispatcher_object; !(k$c_timer);
) RETURNS boolean;
```

**Parameters:**
- `timer`: The virtual address of a dispatcher object of type `timer`.

If the specified timer is currently in the timer queue, then it is removed and a function value of `TRUE` is returned. Otherwise no operation is performed and a function value of `FALSE` is returned.

The Signal state of a timer can be read by calling a kernel function that has the following declaration:

```
PROCEDURE k$tread_timer (
    IN timer : POINTER k$dispatcher_object; !(k$c_timer);
) RETURNS boolean;
```

**Parameters:**
- `timer`: The virtual address of a dispatcher object of type `timer`.

The Signal state of the timer is returned as the function value.

### 4.8.1.6 Thread Object

A thread object is the agent that executes program code and is dispatched for execution by the kernel. This section describes thread objects, and briefly discusses some related topics such as dispatcher states and software priority.

Each thread is associated with a process object. The process object specifies mapping information for the thread and accumulates thread execution time. A thread executes in kernel and user mode, usually at IPL 0, and is dispatched for execution according to its software priority by the thread dispatcher. Several threads can be associated with the same process object.

Although there is only one type of thread object, some threads are referred to by special names according to the function they perform. Function processor, or driver, threads are responsible for controlling hardware devices or presenting the abstraction of a device to the next higher level of software. These threads never execute in user mode and generally do not take page faults. Closely related threads that perform system housekeeping activities are called system threads. These threads may or may not take page faults, according to their individual functions, and also do not execute in user mode. The remaining class of threads are those that execute user software, mostly in user mode. These threads are referred to as process threads.

The context of a thread consists of:
- Scalar registers R2 through R63
- A kernel stack pointer
- A user stack pointer (process threads only)
- Vector registers V0 through V15 (process threads only)
- AST summary and enable registers
- The accumulated cycle count

4–18 The Kernel
All threads have an affinity set that determines on which processor they can execute. This set has a true member for each processor on which the thread can execute. Normally, any thread can execute on any processor.

A thread can be in one of five dispatcher states:

- Ready
- Running
- Standby
- Waiting
- Terminated

A thread is in a Ready state if it can be selected to run on a processor. A ready thread can be in one of the dispatcher ready queues or in a process ready queue.

The state of a thread is Running when it is being executed by a processor.

A thread is in a Standby state when it has been selected to run on a processor, but a context switch to the thread has not actually occurred.

A thread is in a Waiting state when it is waiting for one or more dispatcher objects to attain a Signaled state.

A thread is in a Terminated state after having executed the kernel Terminate Thread procedure.

The process of which the thread is a member can be in an Active or Inactive state (in or out of the balance set). When a process is active, threads in the Ready state are in the dispatcher ready queues. When a process is inactive, threads in the Ready state can be in either one of the dispatcher ready queues or in the process ready queue.

Possible thread state transitions are shown in Figure 4–2.
A set of threads can be logically connected together into a so-called parallel thread set. When a member of a parallel thread set is dispatched, an attempt is made to also dispatch other members of the set. This is done by artificially raising the priority of other members of the thread set which are ready to run to the level of the set member that is being dispatched. If raising the priority does not cause other members to be dispatched, then their priority is not changed and they are not dispatched. A count of the numbers of threads that are simultaneously executing in a process is kept in a user-readable location in the process control region.

A thread is dispatched for execution according to its software priority. Higher-priority threads are given preference and preempt the execution of lower-priority threads. The priority of each thread is divided into a major priority level and a minor priority level. The major priority of a thread is generally controlled by the kernel (for example, priority increments are applied to the major priority), whereas the minor priority can be explicitly modified in order to give preference within a major priority level.

The major and minor priority levels of a thread are combined to form a single continuum of priority from 0, the lowest level, to 63, the highest level. This combined level is simply referred to as the thread priority.

As a thread executes, accounting information is accumulated for the thread, for the thread's process, for the scheduling class of the thread's process, and for the job of the thread's process.
A thread object can be initialized and readied for execution by calling a kernel procedure that has the following declaration:

```c
PROCEDURE
  k$initialize_thread (  
    IN thread : POINTER k$dispatcher_object; !(k$c_thread);  
    IN virtual_stack : POINTER k$context_frame;  
    IN affinity : k$affinity;  
    IN priority : k$combined_priority;  
    IN start_routine : k$start_routine;  
    IN start_context : POINTER anytype;  
    IN queue : POINTER k$dispatcher_object; !(k$c_queue);  
    IN timer : POINTER k$dispatcher_object; !(k$c_timer);  
    IN physical_hwpcb : k$physical_address;  
    IN initial_stack : POINTER k$context_frame;  
    IN kernel_environment : POINTER anytype;  
    IN process : POINTER k$control_object; !(k$c_process);  
  );
```

**Parameters:**

- **thread**: The virtual address of a dispatcher object of type thread.
- **virtual_stack**: The virtual address in the creating process' address space of a kernel stack that is to be used to initialize the thread context frame.
- **affinity**: The initial affinity of the thread.
- **priority**: The combined priority of the thread.
- **start_routine**: A procedure that is to be called the first time the thread is scheduled for execution.
- **start_context**: The virtual address of an arbitrary data structure that is to be passed to the initial thread procedure as a parameter.
- **queue**: The virtual address of a dispatcher object of type queue. The **k$terminate_thread** procedure places the thread in this queue when the thread is terminated.
- **timer**: The virtual address of a dispatcher object of type timer that will be used exclusively by the kernel to implement wait timeout.
- **physical_hwpcb**: The physical address of the hardware privileged context block (HWPCB) within the thread object.
- **initial_stack**: The virtual address of the initial kernel stack in the address space of the process of which the thread is a member.
- **kernel_environment**: A pointer to an arbitrary data structure. The pointer will be stored at a fixed offset from the beginning of the hardware privileged context block (HWPCB), and loaded into R3 when the thread is started.
- **process**: The virtual address of a process object.

The thread object is initialized as a member of the specified process and an initial context frame is built on the kernel stack specified by the **virtual_stack** parameter. The hardware privileged context block (HWPCB) is initialized using the page table base register (PTBR) from the specified process and the kernel stack pointer specified by the **initial_stack** parameter. The **priority** and processor **affinity** are set to their specified values, the initial AST enable register (ASTEN) is set to 1, the initial AST summary register (ASTSR) is cleared, and the thread is readied for execution.

The declaration for the procedure type **k$start_routine** is:

```c
TYPE
  k$start_routine :
    PROCEDURE (  
      IN start_context : POINTER anytype;  
      OUT user_routine : k$user_routine;  
      OUT user_stack : POINTER k$context_frame;  
      OUT user_context : POINTER anytype;  
      OUT user_environment : POINTER anytype;  
    );
```

---

**The Kernel 4-21**
Parameters:

- **start_context**: The virtual address of an arbitrary data structure that is the initial context of the thread.
- **user_routine**: A procedure that is to be called in user mode.
- **user_stack**: The virtual address of a user stack.
- **user_context**: The virtual address of an arbitrary data structure that is to be passed to the user mode procedure as a parameter.
- **user_environment**: The virtual address of the user mode thread environment block.

A thread begins execution in the kernel thread start-up routine which calls the routine specified by the **start_routine** parameter. The **start_routine** is responsible for establishing initial context, and then either calling the actual thread entry point if the thread is a kernel thread, or alternatively returning the user parameters, if the thread is a user thread. If a return to the kernel thread start-up routine is executed, then the user stack pointer is established, an appropriate context frame is built on the user stack, the **user_environment** is loaded into R3, control is transfered to user mode, and the routine specified by the **user_routine** parameter is called with the argument specified by the **user_context** parameter.

The declaration for the procedure type **k$user_routine** is:

```plaintext
TYPE
  k$user_routine : PROCEDURE (
    IN user_context : POINTER anytype;
  );
```

Parameters:

- **user_context**: The virtual address of an arbitrary data structure that is the initial context of the thread.

Waiting on a thread suspends the execution of the invoking thread until the specified thread terminates.

Normally, the affinity of a thread is only set when the thread data structure is initialized. There are cases, however, where it is advantageous to lock the execution of a thread to the current processor or to establish new thread affinity (for example, vector register usage, the zero page thread, and so on).

The kernel function to lock the execution of the current thread on the current processor has the following declaration:

```plaintext
PROCEDURE
  k$lock_execution_thread ( ) RETURNS k$saffinity;
```

The execution of the current thread is locked on the current processor. The previous thread affinity is returned as the function value and may be used to restore the original affinity at a later time.

A kernel function that has the following declaration can be used to define a new thread affinity:

```plaintext
PROCEDURE
  k$set_affinity_thread ( 
    IN affinity : k$saffinity;
  ) RETURNS k$saffinity;
```

Parameters:

- **affinity**: The new affinity of the current thread.

If the thread is using vector instructions, then the specified affinity set is reduced to include only those processors that contain a vector unit.

4-22  The Kernel
If the specified, possibly reduced, affinity set is null, then an error condition is raised. Otherwise the set of processors on which the current thread can execute is set to a new value, and the previous affinity is returned as the function value. If the thread is currently executing on a processor that is not included in the set, an attempt is made to dispatch the thread on a processor that is included in the set, and the current processor is redispached. If the thread cannot be dispatched on a processor that is in the new set, then it is inserted in the dispatcher ready queue selected by its priority.

The combined priority (major and minor) of a thread can be changed by calling a kernel procedure that has the following declaration:

```
PROCEDURE
k$set_combined_priority_thread (  
    IN thread : POINTER k$dispatcher_object; !(k$c_thread);  
    IN priority : k$combined_priority;  
);
```

Parameters:

- **thread**: The virtual address of a dispatcher object of type thread.
- **priority**: The new combined priority of the thread.

The combined priority of the specified thread is set to the new priority value.

The major priority of a thread can be changed by calling a kernel procedure that has the following declaration:

```
PROCEDURE
k$set_major_priority_thread (  
    IN thread : POINTER k$dispatcher_object; !(k$c_thread);  
    IN priority : k$major_priority;  
);
```

Parameters:

- **thread**: The virtual address of a dispatcher object of type thread.
- **priority**: The new major priority of the thread.

The major priority of the specified thread is set to the new priority value. The minor priority is unaffected.

The minor priority of a thread can be changed by calling a kernel procedure that has the following declaration:

```
PROCEDURE
k$set_minor_priority_thread (  
    IN thread : POINTER k$dispatcher_object; !(k$c_thread);  
    IN priority : k$minor_priority;  
);
```

Parameters:

- **thread**: The virtual address of a dispatcher object of type thread.
- **priority**: The new minor priority of the thread.

The minor priority of the specified thread is set to the new priority value. The major priority is unaffected.

The execution of the current thread can be terminated by calling a kernel procedure. The thread will no longer be considered as a candidate for execution. The kernel procedure has the following declaration:

```
PROCEDURE
k$terminate_thread (  
    IN increment : k$priority_increment;  
);
```

The Kernel 4–23
Parameters:

increment  The priority increment that is to be applied if terminating the thread causes a Wait condition to be satisfied.

The current thread object is inserted in the termination queue that was specified when the thread was initialized. It is the responsibility of executive software to process this queue and perform any cleanup operations that are necessary. If the thread has a vector register save area (that is, the vector registers have been used by the thread), the number of the processor on which the thread last used the vector registers is used to locate that processor's processor control block (PB). If the terminating thread was the last thread to use the vector registers on that processor, then pb$last_thread is cleared in that processor's PB.

When a thread is terminated, a new thread is selected for execution on the current processor. If no thread is available, or cannot be executed on the current processor, then the idle thread is executed.

The Signal state of a thread can be read by calling a kernel function that has the following declaration:

```plaintext
PROCEDURE
  k$read_thread (  
      IN thread : POINTER k$dispatcher_object; !(k$s_thread);  
) RETURNS boolean;
```

Parameters:

thread  The virtual address of a dispatcher object of type thread.

The Signal state of the thread is returned as the function value.

### 4.8.1.6.1 Thread Accounting Information

As a thread executes, the number of processor cycles consumed by the thread are accumulated in a cycle count internal processor register which is part of the thread's hardware privileged context block (HWPCB).

The number of cycles a thread has consumed can be determined by calling a kernel function that has the following declaration:

```plaintext
PROCEDURE  
  k$get_cycle_count_thread (  
      IN thread : POINTER k$dispatcher_object; !(k$s_thread);  
) RETURNS large_integer;
```

Parameters:

thread  The virtual address of a dispatcher object of type thread.

The number of cycles consumed by the thread is read from the specified thread's HWPCB and returned as the function value. If the specified thread is currently executing on another processor, then the returned value does not include any time that has been accumulated since the last context switch to the thread.

### 4.8.1.7 Dispatcher Object State Change Synchronization

The kernel provides the capability to synchronize dispatcher object state changes via the object-specific procedures described above and the generic Wait functions. Object-specific procedures notify the kernel of a change in the state of a dispatcher object that may be pertinent to the execution of a thread. Wait functions suspend the execution of a thread until a dispatcher object, or set of dispatcher objects, reach a Signaled state. A queue object, for example, can be empty or contain one or more entries. When the queue contains one or more entries, it is said to be in the Signaled state. If a thread wanted to suspend its execution until the queue contained at least one entry, then it would simply wait on the queue object.
4.8.1.7.1 Wait Block

Threads synchronize their access to other dispatcher objects by executing object-specific procedures and the generic Wait routines. When a thread desires to wait until a dispatcher object reaches a Signaled state, it calls one of the kernel Wait functions. The kernel constructs a data structure called a wait block that describes the dispatcher object to be waited on and the type of Wait. A wait block can be thought of as a surrogate waiter for a thread. The wait block is placed in the wait queue for the respective dispatcher object, and the state of the thread is changed to Waiting.

A wait block has the following type declaration:

```plaintext
TYPE
  k$wait : RECORD
    wa$entry : k$list_entry;
    wa$next : POINTER k$wait;
    wa$thread : POINTER k$dispatcher_object; !(k$c_thread);
    wa$object : POINTER k$dispatcher_object;
    wa$type : k$wait_type[... SIZE (byte)];
    wa$number : integer[0..ord("7F"x)] SIZE (byte);
END RECORD;
```

The fields in the wait block are used as follows:

- `wa$entry` The forward and backward links which are used to insert the wait block into the doubly linked dispatcher object wait queue.
- `wa$next` The address of the next wait block which is used to link together the wait blocks for a specific Wait into a circular list.
- `wa$thread` A pointer to a thread object which is used to locate the thread object that is to be awakened when the Wait is satisfied.
- `wa$object` A pointer to a dispatcher object which is used to locate the dispatcher object that is being waited on.
- `wa$type` The type of the wait which can be either `k$c_wait_any` or `k$c_wait_all`.
- `wa$number` The position in the object list of the dispatcher object that corresponds to the wait block.

A thread can wait on multiple dispatcher objects at a time. Each dispatcher object being waited on requires a wait block. In addition, each Wait can specify a timeout value. If the Wait is not satisfied within the amount of time allowed, then the Wait is automatically satisfied.

In order to avoid the overhead of dynamically allocating and deallocating wait blocks, a fixed number of wait blocks are permanently allocated in the thread object. The number of wait blocks in the thread object determines the maximum number of dispatcher objects that can be waited on simultaneously. This number is currently set to eight.

4.8.1.7.2 Wait

A thread can suspend its execution until one or more dispatcher objects reach a state of Signaled, or until a specified interval of time has expired. The condition of the Wait can be such that all of the objects must attain the Signaled state or, alternatively, until any of the objects reaches the Signaled state.

A thread can suspend its execution by calling one of the following kernel functions:

```plaintext
PROCEDURE
  k$wait_multiple (IN kind : k$wait_type;
                   IN reason : k$wait_reason;
                   IN mode : k$processor_mode;
                   IN object_list : POINTER k$dispatcher_object LIST;
                   IN time_out : large_integer OPTIONAL;
               ) RETURNS integer;
```

The Kernel 4–25
k$s$wait\_several
(IN kind : k$s$wait\_type;
IN reason : k$s$wait\_reason;
IN mode : k$s$processor\_mode;
IN object\_list : POINTER k$s$wait\_object\_list;
IN time\_out : large\_integer OPTIONAL;)
) RETURNS integer;

k$s$wait\_single
(IN reason : k$s$wait\_reason;
IN mode : k$s$processor\_mode;
IN object : POINTER k$s$dispatcher\_object;
IN time\_out : large\_integer OPTIONAL;)
) RETURNS integer;

Parameters:
kind
The kind of Wait that is to be performed: k$c$\_wait\_all or k$c$\_wait\_any.

reason
The reason for the Wait.

mode
The least-privileged processor mode for which an AST can interrupt the Wait: k$c$\_kernel or k$c$\_user.

object
The virtual address of a dispatcher object that is the subject of the Wait operation.

object\_list
A list parameter or pointer to a record that contains the number of, and pointers to, the dispatcher objects that are the subject of the Wait operation.

time\_out
The amount of time in 100 nanosecond units that can expire before the Wait is automatically satisfied.

The function k$s$wait\_multiple is used when it is desirable to wait for more than one dispatcher object, whereas k$s$wait\_single is used when only one dispatcher object is to be waited for. The k$s$wait\_several procedure operates just like k$s$wait\_multiple, except that it accepts the object\_list as a pointer to a record instead of as a LIST parameter.

If the kind of Wait is specified as k$c$\_wait\_any, the Wait function waits until any of the specified dispatcher objects reaches a Signaled state, or until the timeout occurs. If the kind of Wait is specified as k$c$\_wait\_all, then the Wait function waits until all of the specified dispatcher objects simultaneously reach a Signaled state, or until the timeout occurs.

\ k$c$\_wait\_all can be used to simultaneously acquire several mutexes. This is an effective way to guarantee deadlock-free code. However, it is not sufficient in all cases, since quite often mutexes cannot all be acquired at the same time. Even though several mutexes are acquired in a single Wait, they can be released in any order. \n
The reason for the Wait is used to maintain both a Wait state value in the thread object and to keep track of the number of threads that are waiting for various reasons. This parameter is an enumerated type which is defined for use by executive software. The various values are the reasons why the executive would want to put a thread in a Wait state. For example, a particular reason might be associated with a page-fault Wait, whereas a user program executing a user-level Wait procedure might be given a reason of miscellaneous Wait.

The mode of a Wait determines the least-privileged processor mode for which an AST can interrupt the Wait operation. Thus, if the mode value is k$c$\_user, the Wait can be interrupted to deliver either user-mode or kernel-mode ASTs. If, however, the mode value is k$c$\_kernel, then the Wait can only be interrupted to deliver kernel-mode ASTs.

When a thread executes a Wait function, the object\_list is scanned and for each dispatcher object that is specified, a wait block is initialized in the thread object. If a timeout value is also specified, then a timer object is initialized (a timer object for Waits is contained in the thread object) and a wait block is initialized for the timer. The wait blocks are linked together in a circular list and the position of the dispatcher object in the object list is stored in the wait block as the argument number (the position value for k$s$wait\_single is one). The argument number for the timer argument is 0. The address of the current thread object and Wait type (k$c$\_wait\_any or k$c$\_wait\_all) are also stored in the wait block.

4–26 The Kernel
The same dispatcher object should not be specified more than once in the object list. It is the responsibility of executive software to ensure that only meaningful Wait operations are specified (for example, specifying a semaphore twice in a wait all operation will result in the Wait being incorrectly satisfied when the semaphore count is 1).

If any of the dispatcher objects to be waited on is a mutex, then its level number must be greater than the level number of any mutex already owned by the thread or the thread must already own the mutex. (A list, anchored in the thread object, links together all mutexes owned by a thread. This list is in level-number order, such that as mutexes are released, previous level numbers can be calculated.) This guarantees deadlock-free operation with minimum overhead. If a mutex with an equal or lesser level number is specified and the thread does not already own the mutex, then an error condition is raised.

As each dispatcher object is processed and the corresponding wait block is initialized, a check is made to determine if the Wait can be immediately satisfied. If the Wait is for any of the specified objects, then any object that is in a Signaled state will complete the Wait. If the Wait is for all of the specified objects, then all of the objects must be in a Signaled state for the Wait to be completed.

Mutexes are treated as a special case. If one or more of the specified dispatcher objects is a mutex, and the state of the mutex is Not Signaled, then the Wait can still be satisfied if the owner of the mutex is the current thread (recursive acquiring of mutexes within a single thread is allowed).

If the Wait can be immediately satisfied, the thread will not actually enter a Wait state, but rather, will immediately return with a function value equal to the position in the object list of the dispatcher object that satisfied the Wait. If the Wait was for all of the specified objects, an object-specific Wait Satisfied routine must be executed for each of the objects specified in the wait block list (except timeout). If the Wait was for any of the specified objects, then an object-specific Wait Satisfied routine is executed only for the dispatcher object that actually satisfied the Wait (again excepting timeout).

If the Wait is not satisfied, then the wait blocks must be inserted in the wait queue of the respective dispatcher objects they describe. The timer queue entry, if any, is inserted in the timer queue, the current thread is placed in a Wait state, the count of the number of waiting threads for the specified reason is incremented, and the processor is redispached.

If an explicit timeout value of zero is specified and the Wait is not satisfied, the Wait completes immediately with a function value of zero (that is, the value normally returned when a timeout occurs). This feature can be used to synchronously test if the Wait will be satisfied without actually ever entering a Wait state (for example, to conditionally acquire a mutex if it is currently in a Signaled state).

When a thread is placed in a Wait state, its quantum is adjusted. If the adjusted quantum is less than or equal to zero, then quantum end occurs (see Section 4.12.3).

Placing a thread in a Wait state consists of contracting a suitable PC/PS pair, pushing them on the kernel stack, and changing the state of the thread to Waiting. The PC/PS pair is constructed in such a way as to resume execution of the thread in the kernel Wait function with vector instructions disabled (that is, with PS$VEN = 0).

The following steps are executed to place a thread in a Wait state:

1. A new thread is selected for execution on the current processor.

2. The scalar registers are pushed on the kernel stack, and if the Vector Enable bit is set in the current processor status (PS), the vector registers are saved in the vector register save area.

3. The bit corresponding to the current processor number is cleared in the active processor set of the current thread's process object.

4. The headway performance data is updated for voluntary waits.

5. The bit corresponding to the current process number is set in the active processor set of the new thread's process object.

6. The privileged context is swapped; see Section 4.9.2.4.
7. The dispatcher database lock is released.
8. The scalar registers are restored from the kernel stack.
9. An REI is executed to resume execution of the new thread.

When the Wait is satisfied and the thread resumes execution, the reason the Wait was satisfied (Wait completion status) is retrieved from the current thread object and returned as the function value of the Wait.

There are two cases where the thread can resume execution when the Wait conditions were not actually satisfied. These cases arise when a thread is removed from a wait state to deliver a normal or special AST.

If the thread was removed from a wait state to deliver either a special AST or a normal AST that executes in kernel mode, the Wait completion status stored in the thread object is k$c_kernel_ast. After executing the kernel-mode AST, control returns to the Wait function, where the Wait completion status is tested, and the Wait repeated.

If the thread was removed from a wait state to deliver a normal AST that executes in user mode, then the Wait completion status stored in the thread object is k$c_user_ast. In this case, the kernel Wait code must raise the condition deliver_user_ast. This gives executive procedures a chance to clean up data structures and so forth, before actually delivering the user AST. Note that one of the executive condition handlers must do an unwind and execute an REI back to user mode before the AST will actually be delivered (see Section 4.10 for a discussion of AST processing).

4.8.1.7.2.1 Wait Test

Wait Test is an internal kernel procedure that determines if any Wait can be satisfied for a dispatcher object which has a nonempty wait queue and whose state has just transitioned to Signaled. Wait Test is called with the address of the dispatcher object and the priority increment that is to be used to boost the priority of any thread whose Wait can be satisfied.

Wait Test scans the wait queue of the dispatcher object. For each wait block in the wait queue, the circular list of wait blocks is scanned to determine if the wait can be satisfied.

If the complete circular list of wait blocks is scanned and the wait cannot be satisfied, then the scan of the wait queue continues with the next wait block.

If, however, the wait can be satisfied, then the kernel procedure Wait Satisfy is called, specifying the address of the wait block that completed the Wait; see Section 4.8.1.7.2.2. After having satisfied the Wait, the kernel procedure Unwait is called, specifying the argument number and wait block that completed the Wait; see Section 4.8.1.7.2.3. The wait block list is then scanned and each wait block is removed from its respective dispatcher object wait queue.

If the signal state of the object is still Signaled, the scan of the wait queue continues. If, however, the signal state is Not-Signaled, then no further Waits can be satisfied.

The net effect of Wait Test is to scan the wait queue for a dispatcher object that has just reached a Signaled state, trying to satisfy as many Waits as possible.
4.8.1.7.2.2 Wait Satisfy

Wait Satisfy is an internal kernel procedure that satisfies a single Wait. It is called with the address of the wait block that completed the Wait as an argument. Its function is to scan the wait block list, retrieve the respective dispatcher object address, and execute an object-specific routine if satisfying the Wait causes the state of the object to be altered (that is, synchronization event, mutex, or semaphore).

If the wait was for any of the specified objects, only the object-specific routine for the specified wait block is executed. If the wait was for all of the specified objects, then the object-specific routine is executed for each of the dispatcher objects specified by the wait block list.

Note that the object-specific routine may change the state of the dispatcher object to Not-Signaled.

The object-specific routine for mutexes performs several other functions besides satisfying the Wait and changing the Signaled state of the mutex to Not-Signaled. If the mutex is already owned by the thread, then the mutex count is simply decremented. Otherwise, the address of the thread object of the acquiring thread is stored in the mutex, the mutex count is set to 0, and the mutex is inserted in the thread's owned mutex list.

Acquiring a mutex automatically disables normal ASTs that execute in kernel mode (see Section 4.10). It also increments a count in the process object that prevents the thread's process from leaving the balanced set. This ensures that executive code need have no explicit synchronization between non-AST-level code and code that is executed in kernel mode as the result of a normal AST.

When a thread acquires its first mutex, its current priority is saved, then its priority is raised to the maximum of the thread's current priority and the priority of the lowest real-time priority class if priority increments are enabled for the thread's process. The thread is allowed to execute with this possibly elevated priority until it no longer owns any mutexes.

4.8.1.7.2.3 Unwait

Unwait is an internal kernel procedure that returns a waiting thread to a Ready state. It is called with the Unwait status, the priority increment that is to be applied, and the address of the starting wait block. The Unwait status is stored in the target thread's thread object. It is either the argument number of the wait block that completed the Wait or a status that indicates the thread was unwaited to deliver an AST.

Unwait decrements the count of the number of threads waiting for the particular reason, sets the thread state to Ready, and determines into which queue the thread should be inserted.

If the thread does not execute in one of the real-time priority classes and its process does not specify that priority increments are disabled, then its new priority is calculated by adding the specified priority increment to the base priority of the thread's process. If the resultant priority is less than the thread's previous priority (priority when it entered the Wait state), then the priority is not changed. Otherwise, the priority is raised to the calculated value.

4.8.1.7.3 Wait Performance Data

The kernel Wait functions accept an argument which is the reason for the Wait. The value of this argument is used to index an array that records the number of threads that are waiting for each of the possible Wait reasons. When a thread enters a Wait state, the Wait reason is recorded in the thread object and the appropriate element in the Wait array is incremented. Later, when the Wait is satisfied, the original reason for the Wait is retrieved from the thread object and used to decrement the appropriate element in the reason array. In this manner, the total number of threads that are in a Wait state and the number that are waiting for each of the possible reasons can be calculated. Also, the number of threads that are currently waiting for each dispatcher object is maintained by the Wait routines. This data can be read via performance data collection procedures; see Section 4.16.
4.8.2 Control Objects

A control object is a kernel object that is used to direct the operation of the kernel and to control processor execution. Control objects differ from dispatcher objects in that they are not used for synchronization, cannot be waited on, and do not have a state.

Control objects are used to direct the kernel to perform some particular action, such as connecting to an interrupt or the insertion of an entry in a VAX port queue.

The following control objects are supported by the kernel:

<table>
<thead>
<tr>
<th>Control Object</th>
<th>Type Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>AST</td>
<td>k$control_object(k$c_ast)</td>
</tr>
<tr>
<td>Device work queue</td>
<td>k$control_object(k$c_work_queue)</td>
</tr>
<tr>
<td>Interrupt</td>
<td>k$control_object(k$c_interrupt)</td>
</tr>
<tr>
<td>Power-up request</td>
<td>k$control_object(k$c_power_request)</td>
</tr>
<tr>
<td>Power-up status</td>
<td>k$control_object(k$c_power_status)</td>
</tr>
<tr>
<td>Process</td>
<td>k$control_object(k$c_process)</td>
</tr>
<tr>
<td>VAX port queue</td>
<td>k$control_object(k$c_vax_queue)</td>
</tr>
</tbody>
</table>

4.8.2.1 AST Object

An AST object provides the capability to interrupt the execution of a thread and cause a procedure(s) to be called at a specified processor mode. Such an interruption is called an asynchronous system trap (AST). Software running in kernel mode can be interrupted to execute procedures in kernel mode, whereas software running in user mode can be interrupted to execute procedures in either kernel or user mode.

There are two types of AST objects:

1. Special
2. Normal

Special AST objects cause the execution of a thread to be interrupted to execute a procedure in kernel mode at IPL 1 which causes further ASTs to be disabled (the AST interrupt occurs at IPL 1). Normal AST objects cause the execution of a thread to be interrupted to execute a procedure in kernel mode at IPL 1 and, in addition, a procedure in either kernel or user mode at IPL 0. The procedure executed in kernel mode at IPL 1 is called just prior to executing the procedure in the specified mode. When a normal AST is active for a particular processor mode, further normal ASTs are software disabled for that mode until an exit from the AST procedure is executed. Special ASTs, however, interrupt the execution of kernel-mode normal ASTs. See Section 4.10 for more information on AST processing.

An AST object is initialized by calling a kernel procedure that has the following declaration:

```c
PROCEDURE
  k$initialize_ast (    
    IN ast : POINTER k$control_object; !(k$c_ast);   
    IN kernel_routine : k$kernel_ast_routine;    
    IN mode : k$processor_mode OPTIONAL;           
    IN normal_routine : k$normal_ast_routine OPTIONAL;   
    IN context : POINTER anytype OPTIONAL;       
  );
```
Parameters:

ast The virtual address of a control object of type AST.
kernel_routine The procedure that is to be executed at IPL 1 in kernel mode.
mode The processor mode in which the normal AST routine is to be executed: k$c_kernel or k$c_user.
normal_routine The procedure that is to be executed at IPL 0 in the specified processor mode.
context The virtual address of an arbitrary data structure that will be passed to the specified normal AST routine as a parameter.

The declarations for the procedure types $k$kernel.astroutine and $k$normal.astroutine are:

```plaintext
TYPE
  $k$kernel.astroutine :
    PROCEDURE (  
      IN ast : POINTER k$control_object; !(k$c.ast);
    );

  $k$normal.astroutine :
    PROCEDURE (  
      IN context : POINTER anytype;
      IN system_value : quadword CONFORM;
    );
```

Parameters:

ast The virtual address of a control object of type AST.
context The virtual address of an arbitrary data structure.
system_value A quadword value that is provided as an input parameter when an AST object is queued.

The type of AST object to be initialized is determined by the presence or absence of the optional parameters. If any of the optional parameters are specified, then all of them must be specified and a normal AST object is initialized. Otherwise a special AST object is initialized.

The kernel_routine parameter specifies the procedure that will be executed in kernel mode at IPL 1. This routine is called with a single parameter which is the virtual address of the AST object itself.

The mode parameter specifies the processor mode in which the procedure specified by the normal_routine parameter is to be executed. The normal_routine procedure is called with two arguments: one specified by the context parameter, and the other specified by the system_value parameter which is specified when the AST object is inserted in an AST queue (see below).

In order to actually interrupt the execution of a thread, an AST object must be initialized, then inserted in the target thread's AST queue. When proper enabling conditions are present, execution of the thread is interrupted, its context saved, and the specified procedure(s) executed.

The kernel function to insert an AST object into a thread's AST queue has the following declaration:

```plaintext
PROCEDURE
  $k$insert.ast.queue (  
    IN ast : POINTER k$control_object; !(k$c.ast);
    IN system_value : quadword CONFORM OPTIONAL,
    IN thread : POINTER k$dispatcher.object; !(k$c_thread);
    IN wait : boolean = false;
  ) RETURNS boolean;
```

Parameters:
The virtual address of a control object of type AST.

A quadword value that is to be passed to the normal AST routine as a parameter.

The virtual address of a dispatcher object of type thread.

A boolean value that specifies whether the call to the $k$insert ASTM queue function will be immediately followed by a call to one of the Wait functions.

If the value of the wait parameter is TRUE, then the call to the $k$insert ASTM queue function MUST be IMMEDIATELY followed by a call to one of the Wait functions, since a return to the caller is executed at IPL 2 with the dispatcher database locked. This capability can be used to ensure that a context switch does not occur between the insertion in an ASTM queue and the subsequent Wait operation.

If the specified ASTM object is already in an ASTM queue, no operation is performed, and a function value of FALSE is returned. Otherwise, the ASTM active bit is set in the ASTM object, and the ASTM object is inserted in either the kernel-mode or user-mode ASTM queue of the specified thread. An ASTM interrupt is then requested for the target thread by writing its ASTRR register, and a function value of TRUE is returned.

When proper enabling conditions are present, an ASTM interrupt occurs in the target thread. The ASTM object is removed from its respective ASTM queue, the ASTM active bit is cleared, and the procedure(s) specified in the ASTM object are called in the proper mode(s).

An exit from the normal ASTM currently active for user mode can be performed by calling a kernel procedure that has the following declaration:

```
PROCEDURE
  $k$exit ASTM (
    ) RETURNS boolean;
```

If an ASTM is currently active for user mode, then TRUE is returned as the function value. Otherwise FALSE value is returned.

If an ASTM is currently active, then the ASTM active bit for user mode is cleared. Further ASTMs for user mode can now occur. If there are one or more outstanding ASTMs for user mode when this procedure is called, then the an ASTM interrupt is requested by writing ASTRR with a value of $k$c_user.

The exit from a normal ASTM currently active for kernel mode is automatically performed when the normal ASTM procedure returns to its caller. If there are one or more outstanding ASTMs for kernel mode when the normal ASTM routine returns, then the next ASTM for kernel mode is delivered immediately.

An ASTM object can be removed from a thread's ASTM queue by calling a kernel procedure that has the following declaration:

```
PROCEDURE
  $k$remove ASTM queue (
    IN ast : POINTER $k$control_object; !(k$c$ast);
  );
```

Parameters:

ast

The virtual address of a control object of type AST.

If the ASTM object is not in an ASTM queue (active bit is clear), no operation is performed. Otherwise, the specified ASTM object is removed from its ASTM queue, and the active bit is cleared.

The entire kernel or user ASTM queue for a specified thread can be flushed by calling a kernel procedure that has the following declaration:

```
PROCEDURE
  $k$flush ASTM queue (
    IN thread : POINTER $k$dispatcher_object; !(k$c$thread);
    IN mode : $k$processor_mode;
  ) RETURNS POINTER $k$list entry;
```
Parameters:

thread The virtual address of a dispatcher object of type thread.
mode The processor mode for which the AST queue is to be flushed.

The specified AST queue is flushed by removing the list of AST objects from the AST queue header in the thread object. The header is reinitialized and a pointer to the entry for the first AST object is returned as the function value. If the AST queue is empty, then a null pointer is returned.

It is the responsibility of executive software to scan the list of AST objects and perform whatever action is required.

4.8.2.2 Device Work Queue Object

A device work queue object is a repository for device work queue entries and is used to communicate between a driver thread and its interrupt service routine. Typically the interrupt service routine inserts an entry in a device work queue and then signals an event. The driver thread waits on the event and removes entries from the queue when awakened.

A device work queue object is initialized by calling a kernel procedure that has the following declaration:

PROCEDURE
  k$initialize_work_queue (  
    IN work_queue : POINTER k$control_object; !(k$c_work_queue);  
    IN ipl : k$ipl;  
  );

Parameters:

work_queue The virtual address of a control object of type work queue.
ipl The highest interrupt priority level (IPL) from which the device work queue will be accessed.

An entry can be inserted at the head or tail of a device work queue object by calling one of the following kernel functions:

PROCEDURE
  k$insert_head_work_queue (  
    IN work_queue : POINTER k$control_object; !(k$c_work_queue);  
    IN entry : POINTER k$list_entry;  
  ) RETURNS boolean;

  k$insert_tail_work_queue (  
    IN work_queue : POINTER k$control_object; !(k$c_work_queue);  
    IN entry : POINTER k$list_entry;  
  ) RETURNS boolean;

Parameters:

work_queue The virtual address of a control object of type work queue.
entry The virtual address of the work queue entry that is to be inserted at the head or tail of the work queue.

If the specified entry is inserted into an empty work queue, a function value of TRUE is returned. Otherwise, a function value of FALSE is returned.

An entry can be removed from the head of a device work queue object by calling a kernel function that has the following declaration:

PROCEDURE
  k$remove_head_work_queue (  
    IN work_queue : POINTER k$control_object; !(k$c_work_queue);  
  ) RETURNS POINTER k$list_entry;
Parameters:

work_queue  The virtual address of a control object of type work queue.

If the work queue is empty, a null pointer is returned as the function value. Otherwise, a pointer to the entry that is removed from the work queue is returned.

4.8.2.3 Interrupt Object

An interrupt object provides the capability for a driver thread to connect an interrupt vector in the system control block (SCB) to a device interrupt service routine, or to disconnect such a vector. A kernel procedure is also provided that enables a driver thread to synchronize its execution with the execution of an interrupt service routine.

An interrupt object is initialized by calling a kernel procedure that has the following declaration:

```c
PROCEDURE
  k$initialize_interrupt (
    IN interrupt : POINTER k$control_object; !(k$c_interrupt);
    IN service_routine : k$interrupt_routine;
    IN context : POINTER anytype;
    IN offset : integer;
    IN ipl : k$ipl;
  );
```

Parameters:

interrupt  The virtual address of a control object of type interrupt.

service_routine  The procedure that is to be executed each time an interrupt is generated via the specified system control block (SCB) vector.

context  The virtual address of an arbitrary data structure that is to be passed to the specified interrupt service routine as a parameter.

offset  The offset, in longwords, of the interrupt vector in the system control block (SCB).

ipl  The interrupt priority level of the interrupting source.

The declaration for the procedure type `k$interrupt_routine` is:

```c
TYPE
  k$interrupt_routine :
    PROCEDURE (    
      IN interrupt : POINTER k$control_object; !(k$c_interrupt);
      IN context : POINTER anytype;
    );
```

Parameters:

interrupt  The virtual address of a control object of type interrupt.

context  The virtual address of an arbitrary data structure.

Initializing an interrupt object causes code to be generated that will receive control when an interrupt is generated. This includes code to:

1.  Save the volatile registers (those destroyed by the standard calling sequence) on the current kernel stack.

2.  Increment the count of interrupts that have occurred for the interrupt object.

3.  Acquire the interrupt-specific spin lock.

4.  Call the interrupt service routine, specifying as parameters the address of the interrupt object and the `context` parameter.

5.  Release the interrupt-specific spin lock.

4–34 The Kernel
6. Restore the volatile registers from the current kernel stack.

7. Execute an REI to dismiss the interrupt.

An interrupt object can be connected to a system control block (SCB) vector by calling a kernel function that has the following declaration:

```plaintext
PROCEDURE
    k$sconnect_interru ( IN interrupt : POINTER k$control_object; !(k$c_interrupt);
) RETURNS boolean;
```

Parameter:

```plaintext
interrupt       The virtual address of a control object of type interrupt.
```

If the interrupt object is already connected, no operation is performed, and a function value of FALSE is returned. Otherwise, the interrupt object specified by the object parameter is set active, and it is connected to the appropriate SCB vector. A function value of TRUE is returned.

Once an interrupt object is connected to an SCB vector, the specified interrupt service routine is called each time an interrupt is received from the interrupting source.

An interrupt object can be disconnected from an SCB vector by calling a kernel function that has the following declaration:

```plaintext
PROCEDURE
    k$disconnect_interru ( IN interrupt : POINTER k$control_object; !(k$c_interrupt);
); 
```

Parameters:

```plaintext
interrupt       The virtual address of a control object of type interrupt.
```

If the interrupt object is not connected, no operation is performed. Otherwise, the interrupt object is set inactive, and disconnected from the appropriate SCB vector.

A function processor thread can synchronize its execution with the execution of an interrupt service by calling a kernel function that has the following declaration:

```plaintext
PROCEDURE
    k$synchronize_execution ( IN interrupt : POINTER k$control_object; !(k$c_interrupt);
    IN synch_routine : k$synchronized_routine;
    IN context : POINTER anytype;
) RETURNS boolean;
```

Parameters:

```plaintext
interrupt         The virtual address of a control object of type interrupt.
synch_routine     The function whose execution is to be synchronized with the execution of the interrupt service routine that is associated with the specified interrupt object.
context           The virtual address of an arbitrary data structure that is to be passed as a parameter to the synchronized procedure.
```

The declaration for the procedure type k$synchronized_routine is:

```plaintext
TYPE
    k$synchronized_routine :
        PROCEDURE ( IN context : POINTER anytype;
                  ) RETURNS boolean;
```

The Kernel 4-35
Parameters:

context The virtual address of an arbitrary data structure.

The kernel function raises the IPL to that of the interrupting source and acquires the interrupt-specific spin lock. The function specified by the synch Routine parameter is then called with the argument specified by the context parameter. On return, the interrupt-specific spin lock is released, the IPL is restored to its previous value, and the synch Routine function value is returned as the function value.

4.8.2.3.1 Interrupt Object Performance Data

Performance data is collected for each connected interrupt object to determine the number of interrupts that have been received from the interrupting source and the degree of contention that occurs between the interrupt service routine and device driver code. Contention data is collected by the spin count in the spin lock that is used to synchronize device driver execution with the execution of the interrupt service routine. A count of the number of interrupts that have been received from the interrupt source is incremented at each entry to the interrupt service routine. This data can be read via a performance data collection procedure; see Section 4.16.

4.8.2.4 Power-Up Request Object

A power-up request object provides the capability to request that an AST be queued when a power recovery interrupt is generated by PRISM hardware.

A power-up request object is initialized by calling a kernel procedure that has the following declaration:

PROCEDURE
    k$initialize_power_request (  
        IN power_request : POINTER k$control_object; !(k$control_object);
    );

Parameters:

power_request The virtual address of a control object of type power-up request.

A power-up request object can be inserted in the power-up request queue by calling a kernel procedure that has the following declaration:

PROCEDURE
    k$insert_power_request_queue (  
        IN power_request : POINTER k$control_object; !(k$control_object);
        IN ast : POINTER k$control_object; !(k$control_object);
    ) RETURNS boolean;

Parameters:

power_request The virtual address of a control object of type power-up request.
ast The virtual address of a control object of type AST.

If the power-up request object is already in the power-up request queue, no operation is performed and a function value of FALSE is returned. Otherwise, the specified power-up request object is set active, and it is inserted in the power-up request queue. A function value of TRUE is returned.

The power-up request queue is processed at fork level when the power recovery fork block (pb$power_fork) is inserted in the fork queue in response to the power recovery interrupt; see Section 4.9.3. If a power recovery interrupt occurs, the specified AST object is inserted in the AST queue of the thread that executed the insertion procedure.

A power-up request object, when inserted in the power-up request queue, is a one-shot operation. That is, the AST object is queued exactly once, and the power-up request object is then removed from the power-up request queue and set inactive.
A power-up request object can be removed from the power-up request queue by calling a kernel procedure that has the following declaration:

```
PROCEDURE
  k$s$remove$power$queue$queue$(
    IN power_request : POINTER k$s$control$object; !k$s$power$request);
);
```

Parameters:

- `power_request` The virtual address of a control object of type power-up request.

If the power-up request object is not in the power-up request queue, no operation is performed. Otherwise, the specified power-up request object is removed from the power-up request queue, and set inactive.

### 4.8.2.5 Power-Up Status Object

A **power-up status object** provides the capability to request that a specified variable's value be set to **TRUE** when a power recovery interrupt is generated by PRISM hardware.

A power-up status object is initialized by calling a kernel procedure that has the following declaration:

```
PROCEDURE
  k$s$initialize$power$status$(
    IN power_status : POINTER k$s$control$object; !k$s$power$status);
);
```

Parameters:

- `power_status` The virtual address of a control object of type power-up status.

A power-up status object can be inserted in the power-up status queue by calling a kernel procedure that has the following declaration:

```
PROCEDURE
  k$s$insert$power$status$queue$(
    IN power_status : POINTER k$s$control$object; !k$s$power$status;
    IN state : POINTER boolean;
  ) RETURNS boolean;
```

Parameters:

- `power_status` The virtual address of a control object of type power-up status.
- `state` The virtual address of a state variable that is to be set to a value of **TRUE** if the power fails.

If the power-up status object is already in the power-up status queue, no operation is performed, and a function value of **FALSE** is returned. Otherwise, the specified power-up status object is set active, and the value of the specified variable is set to **FALSE**. The power-up status object is then inserted in the power-up status queue, and a function value of **TRUE** is returned.

The power-up status queue is processed at IPL 7 after having received a power recovery interrupt. If a power recovery interrupt occurs, each entry in the power-up status queue is removed, the power status object is set inactive, and the variable specified by the `state` parameter is set to **TRUE**.

A power-up status object, when inserted in the power-up status queue, is a one-shot operation. That is, the status variable is set to a value of **TRUE** exactly once, and the power-up status object is then removed from the power-up status queue and set inactive.

A power-up status object can be removed from the power-up status queue by calling a kernel procedure that has the following declaration:

```
PROCEDURE
  k$s$remove$power$status$queue$(
    IN power_status : POINTER k$s$control$object; !k$s$power$status);
);
```
Parameters:

\texttt{power\_status} \quad \text{The virtual address of a control object of type power-up status.}

If the power-up status object is not in the power-up status queue, no operation is performed. Otherwise, the specified power-up status object is removed from the power-up status queue, and the object is set inactive.

### 4.8.2.6 Process Object

A process object represents the address space and control information necessary for the execution of a set of thread dispatcher objects. The process object accumulates total execution time and is used by the balance set scheduler to determine which set of processes are active at any point in time.

A process object is initialized by calling a kernel procedure that has the following declaration:

\begin{verbatim}
PROCEDURE k$initialize_process
    IN process : POINTER k$control_object; !(k$c_process);
    IN priority : k$combined_priority;
    IN increment : boolean;
    IN decrement : boolean;
    IN thread_quantum : integer;
    IN process_quantum : integer;
    IN active_count : POINTER integer;
    IN event : POINTER k$dispatcher_object; !(k$c_event);
    IN ptbr : POINTER k$page_frame;
)
\end{verbatim}

Parameters:

- \texttt{process} \quad \text{The virtual address of a control object of type process.}
- \texttt{priority} \quad \text{The base priority of the process.}
- \texttt{increment} \quad \text{A boolean variable that enables (TRUE) or disables (FALSE) priority increments.}
- \texttt{decrement} \quad \text{A boolean variable that enables (TRUE) or disables (FALSE) priority decrements.}
- \texttt{thread_quantum} \quad \text{The amount of time that a thread within the process is allowed to run before its priority is subject to adjustment in one millisecond units.}
- \texttt{process_quantum} \quad \text{The number of thread quantum ends before the process is subject to being removed from the balance set.}
- \texttt{active_count} \quad \text{A pointer to the count of active threads.}
- \texttt{event} \quad \text{The virtual address of a dispatcher object of type event.}
- \texttt{ptbr} \quad \text{The page frame number of the segment 1 page table page.}

The process object is initialized with the base priority specified by the \texttt{priority} parameter. The \texttt{increment} and \texttt{decrement} parameters control whether priority increments and decrements are enabled for threads that are members of the process. The \texttt{thread_quantum} parameter controls the rate at which the priority of threads within the process will decay (decrements must also be enabled). The \texttt{process_quantum} parameter controls the duration of time before the process will be subject to removal from the balance set.

The \texttt{active_count} parameter specifies a pointer to an integer that will record the number of threads simultaneously active for the process. The active count is cleared when the process object is initialized. This pointer can be set to another location using the procedure \texttt{k$set_count_pointer_process}, which is described below. The \texttt{event} parameter specifies an event object that can be used to synchronize the removal of the process from the balance set (see \texttt{k$clear_active_process} below). The \texttt{ptbr} parameter specifies the page frame number of a page that contains the segment 1 page table page for the process.

---

4-38 The Kernel
If priority increments are enabled then a thread is given a priority boost when it exits a wait state, provided the boosted priority is greater than the thread's previous priority and the thread does not execute in a real-time priority class. If priority decrements are enabled, then the thread's priority will decay at quantum end provided the decayed priority is not less than 4 and the thread does not execute in a real-time priority class.

The pointer to the count of active threads can be set to another location by calling a kernel function that has the following declaration:

```
PROCEDURE
    k$set_count_pointer_process (  
        IN active_count : POINTER integer;
    );
```

Parameters:

- `active_count`: A pointer to the count of active threads.

The current count of active threads is copied to the `active_count` parameter.

The state of a process can be changed to inactive by calling a kernel function that has the following declaration:

```
PROCEDURE
    k$clear_active_process (  
        IN process : POINTER k$control_object; !(k$c_process);
    ) RETURNS boolean;
```

Parameters:

- `process`: The virtual address of a control object of type `process`.

The state of the specified process is set to inactive. If no thread in the process owns a mutex, the process is an immediate candidate for removal from the balance set, and a function value of `TRUE` is returned. If, however, one or more threads own mutexes, then the process cannot be removed from the balance set until all such threads have released the mutexes they own. This is accomplished by allowing threads that own mutexes to continue to be considered for execution by the thread dispatcher until they have released all mutexes they own. For this case, the event that was specified when the process was initialized is used to determine exactly when the process is inactive. The event is cleared when the `k$clear_active_process` function is executed and set when the last thread releases its last mutex.

As threads are considered for execution (that is, threads in the Ready state), a test is made to determine if their process is inactive. If the thread's process is inactive and the thread does not own any mutexes, it is inserted in the process ready queue. Otherwise, its state is changed to Standby and the selected processor is directed to dispatch. A similar test is made when a thread's state is changed to Ready.

Processors on which threads for the process are running (Running state) or about to run (Standby state) are rescheduled if the respective threads do not own any mutexes.

This procedure is intended for use by the balance set scheduler. Once it has decided to remove a process from the balance set, it simply calls the kernel, specifying the process to be set inactive. If the process contains threads that currently own mutexes, then the balance set scheduler can wait on the process activity event to determine exactly when the process can be removed from the balance set.

The state of a process can be changed to active by calling a kernel procedure that has the following declaration:

```
PROCEDURE
    k$set_active_process (  
        IN process : POINTER k$control_object; !(k$c_process);
    );
```

The Kernel 4-39
Parameters:

process The virtual address of a control object of type process.

The state of the specified process is set to active and the process ready queue is examined. The process ready queue is a list of thread's that attained a ready state while the process was inactive. Each thread in the process ready queue is removed and readied for execution.

This procedure is intended for use by the balance set scheduler. Once it has decided to insert a process in the balance set, it simply calls the kernel, specifying the process to be set active.

The base priority of a process and the current priority of all its active threads can be changed by calling a kernel procedure that has the following declaration:

PROCEDURE
k$set_priority_process (
    IN process : POINTER k$control_object; ! (k$c_process); 
    IN priority : k$combined_priority;
);

Parameters:

process The virtual address of a control object of type process.
priority The new base priority of the process.

The base priority of the process is set to the specified value. The list of threads associated with the process is then examined to determine their new priorities.

If the new base priority of the process is above 47 (that is, a real-time priority class), the combined priority of each thread is set to the base priority of the process. If the new base priority is less than 48 (that is, a class scheduling priority), then the new priority of each thread is computed by adding the difference between the new base priority and the old base priority to the thread priority. This value is not allowed to go above 47 or below 4.

4.8.2.6.1 Process Accounting Information

As a thread executes, the number of processor cycles consumed by the thread are accumulated in a cycle count internal processor register which is part of the thread's hardware privileged context block (HWPCB).

The aggregate number of processor cycles consumed by all the threads in a process is maintained by summing the incremental cycles used by each thread at each context switch.

The number of cycles consumed by all the threads in a process can be determined by calling a kernel function that has the following declaration:

PROCEDURE
k$get_cycle_count_process ( 
    IN process : POINTER k$control_object; ! (k$c_process); 
) RETURNS large_integer;

Parameters:

process The virtual address of a control object of type process.

The current aggregate number of cycles consumed by all the threads in the specified process is returned as the function value. If any of the threads of the process are currently executing, then the returned value does not include any time that has been accumulated by these threads since the last context switch to the thread.
4.8.2.7 VAX Port Queue Object

A VAX port queue object is a repository for VAX port queue entries and is used to communicate between a PRISM processor and a VAX port device controller. Entries are inserted by either the VAX port device controller or a driver thread running on a PRISM processor in a manner compatible with VAX self-relative interlocked queues.

A VAX port queue object is initialized by calling a kernel procedure that has the following declaration:

PROCEDURE
  k$initialize_vax_queue (  
    IN vax_queue: POINTER k$control_object; !(k$c_vax_queue);  
  );

Parameters:

vax_queue   The virtual address of a control object of type VAX port queue.

VAX port queues are VAX self-relative interlocked queues. The low-order bit of the queue header serves as the interlock for the queue. The queue header stores the relative addresses of the first and last entry in the queue. The first two longwords of a queue entry contain the relative address of the next and previous entries in the queue.

An entry can be inserted at the head or tail of a VAX port queue object by calling one of the following kernel functions:

PROCEDURE
  k$insert_head_vax_queue (  
    IN vax_queue: POINTER k$control_object; !(k$c_vax_queue);  
    IN entry: POINTER k$list_entry;  
  ) RETURNS boolean;

PROCEDURE
  k$insert_tail_vax_queue (  
    IN vax_queue: POINTER k$control_object; !(k$c_vax_queue);  
    IN entry: POINTER k$list_entry;  
  ) RETURNS boolean;

Parameters:

vax_queue   The virtual address of a control object of type VAX port queue.
entry       The virtual address of the entry that is to be inserted at the head or tail of the VAX port queue.

If the entry is being inserted into an empty queue, a function value of TRUE is returned. Otherwise, a function value of FALSE is returned.

An entry can be removed from the head of a VAX port queue object by calling a kernel function that has the following declaration:

PROCEDURE
  k$remove_head_vax_queue (  
    IN vax_queue: POINTER k$control_object; !(k$c_vax_queue);  
  ) RETURNS POINTER k$list_entry;

Parameters:

vax_queue   The virtual address of a control object of type VAX port queue.

If the VAX port queue is empty, a null pointer is returned as the function value. Otherwise, the virtual address of the entry that is removed from the VAX port queue is returned.
4.8.3 Miscellaneous Kernel Routines

The kernel provides several miscellaneous routines that do not manipulate kernel objects, but are essential to the operation and control of the processor.

4.8.3.1 Set Interrupt Priority Level

A driver thread can set the interrupt priority level (IPL) of the current processor by calling a kernel function that has the following declaration:

\[
\text{PROCEDURE}
\begin{align*}
\text{k$set\_interrupt\_priority\_level (} \\
\text{IN ipl : k$ipl;} \\
\text{)} \text{ RETURNS k$ipl;}
\end{align*}
\]

Parameters:

\[
ipl \quad \text{The new interrupt priority level.}
\]

This function returns the previous IPL as the function value and sets the current IPL to the specified value. This capability is intended for use in conjunction with disabling power failure over short spans of code, such as the loading of device registers. It should only be called from an interrupt service routine or from code that is executed at device IPL while holding the interrupt-specific interlock; see Section 4.8.2.3.

Extreme caution must be exercised when using this function. If power failure recognition is prohibited for too long a period and an actual failure of power occurs, then it will not be possible to recover and continue system operation without rebooting when power is finally restored.

4.8.3.2 Set Event From Interrupt

An interrupt service routine can set the state of an event to Signaled by calling a kernel procedure that has the following declaration:

\[
\text{PROCEDURE}
\begin{align*}
\text{k$set\_event\_from\_interrupt (} \\
\text{IN event : POINTER k$dispatcher\_object; !k$dispatcher\_object);} \\
\text{)}
\end{align*}
\]

Parameters:

\[
event \quad \text{The virtual address of a dispatcher object of type event.}
\]

If the specified event is already in the fork queue, no operation is performed. Otherwise, the fork block contained within the event object is set active, inserted in the current processor's fork queue, and a dispatcher interrupt is requested. When proper enabling conditions exist, a dispatch interrupt is generated at IPL 2 and the fork queue is processed; see Section 4.9.3.

4.9 Scheduling

Scheduling refers to the algorithms whereby threads are successively considered for execution on processors. Usually there is a "scheduler" that selects which threads are eligible for execution and a "dispatcher" that chooses exactly which thread to execute next. Schedulers usually include policy and are the province of higher levels of software. Therefore, the kernel does not contain any scheduling code per se, but rather, maintains data that can be used by higher levels of software to support such things as balance set and class scheduling. Dispatching, on the other hand, is the province of the kernel and several databases are maintained to aid the kernel in carrying out this activity.

4-42 The Kernel
The kernel supports 64 levels of thread priority. The highest 16 levels are referred to as real-time priorities and the lowest 48 levels as class priorities. In the real-time levels, a thread executes at a fixed priority and is not subject to requeuing at quantum end. The priority is neither boosted, nor does it decay as the thread executes and leaves a Wait state. A thread in the class levels has a base priority, and its priority varies from the base level as the thread executes and leaves a Wait state.

Thread dispatching is carried out using a preemptive priority algorithm which attempts to keep the highest priority threads in control of the processor(s). If a thread attains a priority that is higher than that of a currently running thread, then the current running thread is preempted. The preempted thread is requeued at the front of the appropriate dispatcher ready queue to await further execution.

Quantum end causes the priority of a class-level thread to decay and go to the end of the next-lower dispatcher ready queue. If the thread executes long enough without entering a Wait state, the priority will eventually decay to 4 (class priorities are never allowed to decay below 4). Each time the thread leaves a Wait state, it is given a boost to a priority level that is above its base level. The actual increment that the thread is given depends on the object that satisfied the Wait.

Priority increments and decrements can be individually enabled and disabled on a per-process basis.

This is much the same as VMS. There are a few exceptions: 1) a thread's priority does not automatically decay when it is dispatched for execution (it will only decay at quantum end), 2) priority level 0 is a usable priority level, 3) a thread's priority is allowed to decay below its base priority but never below 4, which are reserved for operating system threads such as the zero page thread, 4) threads running at real-time priority levels are not requeued at quantum end, 5) when a thread is preempted it is requeued at the front, rather than the end, of the appropriate ready queue.

4.9.1 Dispatcher Database

The kernel maintains several data structures to aid in choosing which threads should be active at any point in time:

- Ready queues
  There are 64 ready queues corresponding to the 64 levels of thread priority. Each of these queues contains a list of the threads that are ready for execution at the respective level.

- Ready summary
  A 64 member set that corresponds to the 64 levels of thread priority. A member of the set is true if the corresponding ready queue is not empty.

- Active summary
  A 64 member set that corresponds to the 64 levels of thread priority. A member of the set is true if a thread at the corresponding priority level is currently executing, or about to be executed, on a processor.

- Active matrix
  The active matrix is a 64 element array of 32 member sets. It is used to record the processors on which threads at the corresponding priority levels are executing. The elements of the array correspond to the 64 levels of thread priority and the members of the sets correspond to the 32 processors that are supported.

- Idle summary
  A 32 member set that corresponds to the 32 processors that are supported. A member of the set is true if the an idle thread is running on the corresponding processor.

- Idle thread
  A system thread that has the special property of only executing a loop that tests if its processor has been assigned a thread to execute.
The limit of 32 processors is not arbitrary. More than 32 processors could be supported with minimal changes, albeit, with a significant degradation in performance. The number 32 was chosen because this is the number of bits in a longword.

4.9.2 Thread Dispatcher

The thread dispatcher executes in response to various events that cause changes in the state of the system. The dispatcher always executes at IPL 2 with the dispatcher database locked. Thus, no other processor can be simultaneously executing the dispatcher at any given point in time.

The dispatcher is distributed in the sense that when a change in system state occurs, the processor that is effecting the change decides whether its own or any other processor's execution should be preempted to run a different thread. If a change is necessary, then a processor and thread are selected, the state of the thread is changed to Standby, and the target processor is interrupted and directed to redispacht.

Each thread has a set that specifies on which processors it can execute. This is referred to as processor affinity.

The dispatcher always attempts to keep the highest-priority threads in execution at any point in time (up to the number of processors in the system). If no thread has processor affinity (that is, all threads can be executed on all processors), it is guaranteed that the highest-priority threads will actually be in execution. However, if any thread has affinity, then it is only guaranteed that the highest-priority thread will be in execution.

Consider the following example: There are two processors in the system (Ap and Bp) and three threads (Ct, Dt, and Et). Thread Ct has a priority of 10 and can execute on any processor; thread Dt has a priority of 8 and can only execute on processor Ap; and thread Et has a priority of 9 and can only execute on processor Bp. Now, suppose that thread Ct is executing on processor Bp, thread Dt is executing on processor Ap, and thread Et is in a Wait state. The dispatcher object that thread Et is waiting for is then signaled, making thread Et runnable. Threads Ct and Et are the highest-priority threads in the system, but thread Et cannot preempt thread Ct on processor Bp. Therefore, no change in the system state will occur (that is, thread Ct will not be moved to processor Ap so that thread Et can execute on processor Bp). If none of the threads in this example had processor affinity, then thread Et would preempt thread Dt on processor Ap, and the two highest-priority threads would be in execution.

The dispatcher is not a single routine, but rather, key decisions are placed at the point where a change in state can occur. The major points at which such a decision must be made are:

- The current thread enters a Wait state and another thread must be selected to execute on the current processor.
- The current thread terminates and another thread must be selected to execute on the current processor.
- The current thread releases the last mutex it owns and its process is inactive.
- The current thread reaches a quantum end and its priority decays to a level that is below that of another thread that is ready to run.
- The current thread sets its affinity to a value such that the thread is no longer allowed to execute on the current processor.
- The process of which a running thread is a member is set inactive (removed from the balance set).
- The priority of a thread is changed to a level different from its previous level.
- A thread that was previously inactive (waiting, belonging to an inactive process, and so forth) makes the transition to a Ready state.
The first five of these cases only affect the activity of the processor on which the change is occurring. The last three cases can affect the activity of any processor in the configuration. The following sections describe each of these cases in detail.

4.9.2.1 Thread Becomes Unrunnable

A thread can become unrunnable because it goes into a Wait state, terminates, its process becomes inactive, its affinity is set to a value such that it can no longer execute on the current processor, or it releases the last mutex it owns and its process is already inactive.

The priority of the unrunnable thread is used to clear the member corresponding to the processor number in the active matrix. If the particular element in the active matrix becomes a null set, then the member corresponding to the priority is cleared in the active summary.

A new thread must be selected for execution on the target processor, the context of the unrunnable thread saved, and the context of the selected thread restored. This is accomplished by scanning the ready summary, starting with the member specified by the priority of the unrunnable thread and continuing downward (to lesser priorities) until a true member is found or the entire set has been scanned. It is only necessary to scan downward because any higher-priority thread that could execute on the target processor would have already been dispatched and run.

If a true member is found, the corresponding ready queue is not empty. It is not sufficient, however, to have found a nonempty ready queue. The ready queue itself must be scanned to find a thread that has an affinity such that it can execute on the target processor.

If a thread is located that can execute on the target processor, it is removed from its ready queue and the ready summary is updated if necessary (if the respective ready queue becomes empty, then the corresponding member of the ready summary is cleared). The priority of the selected thread is used to set the corresponding member in the active summary and to select the element in the active matrix for which the member corresponding to the processor number is to be set.

If there is no thread that can execute on the target processor, then the per-processor idle thread is selected and the member corresponding to the processor number is set in the idle summary.

The privileged context is swapped; see Section 4.9.2.4. The dispatcher database lock is released, scalar registers are restored, and an REI is executed to resume execution of the selected thread.

4.9.2.1.1 Idle Thread

Each processor has an idle thread that it can always execute. The idle thread has a priority that is below that of all other threads. It is not executed unless there is no other thread that can be run on the respective processor.

When the idle thread is active, the member corresponding to the processor number on which it is executing is set in the idle summary. This set is maintained so that quick selection of a processor can be accomplished when there is an idle processor and a thread executable on that processor becomes active.

The idle thread is executed at IPL 2 and continuously tests to determine if its processor has been assigned a new thread to execute. Thus, when a thread is selected to run on an idle processor, it is not necessary to send the target processor an interrupt directing it to redispach. The idle thread will discover a nonzero next thread object address, acquire the dispatcher database, and cause a context switch to the new thread. As the idle thread continuously tests for a new thread to execute, it also checks if any entries have been inserted in the fork queue. If an entry has been inserted in the fork queue, the idle thread removes the entry and executes the specified action.
4.9.2.2 Priority of A Thread Changes

The priority of the current thread can change because of a quantum end. Quantum end causes thread priority to be lowered if priority decrements have not been disabled for the thread's process, the thread's priority is not a real-time priority, and the thread does not currently own any mutexes. The priority of any thread can also be explicitly changed by calling one of the kernel procedures that sets the thread or process priority. An explicit change in priority can result in a lower or higher priority (see Section 4.8.1.6 for information on the kernel routines for setting thread priority).

The exact action taken when the priority of a thread is changed depends on the state of the thread and whether the thread owns any mutexes.

If the thread owns one or more mutexes, then its real priority was saved when the first mutex was acquired. At that time the thread's priority was possibly raised if priority increments were enabled and the thread was not running in a real-time priority class. For this case the new priority replaces the priority that was saved when the thread acquired its first mutex. If the new priority is less than the current priority of the thread, then no further action need be taken. When the thread releases its last mutex, the saved priority (new priority) will be restored. However, if the new priority is also greater than the current thread priority, then the current priority of the thread must also be set to the new priority.

The following cases apply when the thread does not own any mutexes, or the thread owns one or more mutexes and the new priority is greater than the thread's current priority.

If the thread is in the Terminated or Waiting state, the priority of the thread is simply changed and no further action is necessary.

If the thread is in the Ready state and is currently in the process ready queue, then the priority of the thread can likewise be changed and no further action need be taken.

If the thread is in the Ready state and is not in the process ready queue, then the thread is removed from its dispatcher ready queue, the new priority is set, and an attempt is made to dispatch the thread immediately. If the thread cannot be dispatched, it is inserted in the dispatcher ready queue specified by its new priority. If the priority is being raised, then the thread is inserted at the end of the ready queue. Otherwise (priority is being lowered) it is inserted at the front of the ready queue.

If the thread is in the Running state and the processor on which the thread is executing also has a thread in the Standby state (that is, has been selected for execution on the processor, but has not yet been dispatched), then the priority of Running thread can be simply changed. When a context switch to the thread in the Standby state occurs, the Running thread will be again readied for execution.

If the thread is in the Running state and the processor on which the thread is executing does not have a thread in the Standby state, or the thread is in the Standby state, and the new priority is greater than or equal to the old priority, then the priority of the thread is set to the new value and the dispatcher database is updated. The target processor is already running or about to run the highest priority thread it can. Raising the priority of that thread does not cause a change in the scheduling state of the system.

If the thread is in the Running state and the processor on which the thread is executing does not have a thread in the Standby state, or the thread is in the Standby state, and the new priority is less than the old priority, then the priority of the thread is lowered and the dispatcher database is updated. A scan must then be performed to determine if there is another higher-priority thread that should preempt the execution of the subject thread. The scan is performed starting with the old priority of the thread and continuing downward to the new priority plus one. If a higher-priority thread is located, it is selected for execution on the target processor.
4.9.2.3 Thread Makes the Transition to Ready State

A thread makes the transition to a Ready state by either its Wait being satisfied, its process being returned to the active state, or its execution being preempted by a higher priority thread.

When a thread makes the transition to a Ready state, it must be determined if the thread should preempt the execution of another thread, and if so, on which processor. If the thread should not preempt the execution of another thread, then it is inserted in the dispatcher ready queue specified by its priority, and the ready summary is updated.

The first attempt at dispatching the new thread is determined by examining the idle summary. If there is an idle processor available on which the thread can execute, then the state of the thread is set to Standby and the address of the thread object is stored in the target processor's PB. The priority of the new thread is used to set the processor member in the active matrix and in the active summary. There is no need to interrupt the target processor, as the idle thread immediately determines that another thread has been selected and causes it to be dispatched.

If no idle processor can execute the thread, or there is no idle processor available, then the dispatcher database must be examined to determine if any thread's execution should be preempted.

The active matrix is scanned backwards from priority 0 up to, but not including, the priority of the new thread. The active matrix contains the set of processors that are active at each of the 64 priority levels. The active set at each level is simply ANDed with the affinity set of the new thread, and if the result is nonzero, a processor has been found that can be preempted. A processor must then be selected from this set.

The first attempt at selecting a processor from the set is to try the processor that the thread last ran on. If this processor can be selected, then there is a possibility that the processor caches still contain data belonging to the thread. Otherwise the first available processor, starting at member zero, is selected.

If no processor can be found that should be preempted, then the new thread is inserted at the end of the appropriate dispatcher ready queue, and the ready summary is updated.

4.9.2.4 Privileged Context Swapping

A privileged context swap is performed when switching the processor from one thread to the next. The privileged context swap is performed with the SWPCTX instruction, which saves the privileged context of the previous thread, then loads the privileged context of the new thread. This involves saving the internal processor registers KSP and USP and loading the internal processor registers ASTEN, ASTSR, ASN, PTBR, KSP, and USP.

Before executing the SWPCTX instruction, it may be necessary to assign a new address space number (ASN). If the current contents of the page table base register (PTBR) are not equal to the contents of PTBR in the new thread's hardware privileged context block (HWPCB), then a new ASN must be assigned. This is accomplished by incrementing the ASN allocation sequence number in the processor control block (PB) and storing the result in the new thread's HWPCB. If the incremented ASN sequence value is equal to the number of ASNs provided by the host processor, then the ASN sequence number is reset to the value 0 and the entire translation buffer is flushed.

A SWPCTX is then executed to save the privileged context of the previous thread and load the privileged context of the new thread.

The process object contains an active processor set that is used to record processors on which threads belonging to the process are currently in execution. Before swapping the privileged context, this set must be updated for both the previous thread and the new thread. This is accomplished by clearing the member corresponding to the processor in the previous thread's process object and setting the member corresponding to the processor in the new thread's process object. This information is used to determine which processors must receive interprocessor interrupts when kernel procedures to flush translation buffer entries and caches are called.
4.9.3 Fork Dispatcher

The fork dispatcher is a routine that executes in response to the redisplay interrupt at IPL 2 (software interrupt initiated by an MTPR SIRR). Interrupts at this level are initiated when an entry is inserted into an empty processor-specific fork queue (FQ), and when a new thread has been selected for execution on a target processor.

The fork dispatcher saves volatile registers on the kernel stack and then examines the fork queue. The fork queue is a list of "actions" that are to be performed by the fork dispatcher. These actions include:

- Setting an event
- Processing the timer queue
- Performing quantum end processing
- Processing the power-up request queue

Each action is described by a data structure called a fork block (FB) which contains an active bit, queue linkage information, and the address of the service routine for the particular action to be performed. FBs are generally contained within another structure, such as an event.

The fork dispatcher removes entries from the fork queue and executes the indicated action. When the fork queue is empty, pbb$next_thread in the current processor's PB is examined to determine if a context switch to a new thread should be performed.

As the fork dispatcher processes entries from the fork queue, it raise the IPL to 7, removes the next entry from the fork queue, clears the fork queue entry's active bit, lowers the IPL to 2, and then calls the specified action routine with address of the fork queue entry as an argument.

The following sections describe the actions performed for each type of fork queue entry.

4.9.3.1 Set Event

The kernel provides a routine to set an event from an IPL greater than 2 (k$set_event_from_interrupt). This routine takes as an argument the address of an event object and calculates the address of a contained fork block (FB). The fork block is inserted in the current processor's fork queue. This capability is intended for use by device driver interrupt service routines that wish to signal a system thread when an interrupt occurs.

The action routine to set an event performs the following operations:

1. Calculates the address of the event object from the address of the FB.
2. Calls the routine k$set_event, specifying the event object as an argument.
3. Returns control to the fork dispatcher.

4.9.3.2 Timer Queue

Each processor has a timer queue fork block (TQFB), which is allocated in the processor control block (PB). When an interval timer interrupt occurs, the interrupt service routine examines the timer queue to determine if an entry has expired (actually, it only needs to examine the expiration time of the first entry); see Section 4.12. If the first entry in the timer queue has expired, the TQFB is inserted in the processor's fork queue.

The action routine for the timer queue executes the following operations:

1. Acquires the dispatcher database lock.
2. Processes each entry in the timer queue that has expired, removes the timer object from the timer queue, sets its state to Signaled, and if specified, queues an AST.
3. Computes the time at which next entry in the timer queue is due to expire.
4. Releases the dispatcher database lock.
5. Returns control to the fork dispatcher.

4.9.3.3 Quantum End

Each processor has a quantum end fork block (QEBF), which is allocated in the processor control block (PB). When an interval timer interrupt occurs, the interrupt service routine decrements the quantum of the current thread. If the resultant value is less than or equal to zero, then a quantum end has occurred and the QEBF is inserted in the processor's fork queue; see Section 4.12.

The action routine for quantum end performs the following operations:

1. Acquires the dispatcher database lock.
2. Obtains the address of the current thread from the processor block and checks whether quantum end has occurred for the current thread. This check is necessary since a context switch may have been in progress when the interval timer interrupt occurred and the quantum end applied to the previous thread. If quantum end has not occurred, then the dispatcher database lock is released and control is returned to the fork dispatcher.

If quantum end has occurred and the thread owns one or more mutexes, then the thread quantum is reset to one and no other action is performed. When the thread acquired its first mutex its priority was possibly raised and, therefore, should not be altered until the last mutex is released and the original thread priority is restored. As the thread continues to execute it will continuously receive quantum ends at each interval timer interrupt until the last mutex is released and the quantum is replenished.

If a quantum end has occurred and the thread does not own one or more mutexes, then the quantum of the thread's process is decremented. If the resultant value is less than or equal to zero, then the process is a candidate for removal from the balance set. The thread quantum is replenished and, if priority decrements are enabled, the thread's priority is decremented.

3. Releases the dispatcher database lock.
4. Returns control to the fork dispatcher.

4.9.3.4 Power-Up Request Queue

Each processor has a power recovery fork block (PRFB), which is allocated in the processor control block (PB). When a processor receives a power recovery interrupt, the power recovery fork block (PRFB) is inserted in the processor's fork queue.

The action routine for power recovery performs the following actions:

1. Acquires the power-up request lock and scans the power-up request queue. For each entry in the queue, the following actions are performed:
   • Acquires the dispatcher database lock.
   • Removes the next entry from the power-up request queue, clears its active bit, and inserts the specified AST object in the target thread's AST queue. When proper enabling conditions are present, an AST delivery interrupt will occur in the target thread.
   • Releases the dispatcher database lock.
2. Reinitializes the power-up request queue listhead and releases the power-up request lock.
3. Returns control to the fork dispatcher.
4.9.3.5 Fork Dispatcher Exit

The fork dispatcher removes entries from the fork queue and executes the indicated action. When the fork queue is empty, the fork dispatcher examines pb$next_thread in the current processor's PB.

If pb$next_thread is zero, a new thread has not been selected for execution on the processor. The volatile registers are restored from the kernel stack, and an REI is executed to resume the current thread's execution.

If pb$next_thread is not zero, then the dispatcher database lock is acquired and pb$next_thread is reread (it may have changed between being tested and locking of the dispatcher database). The registers not included in the volatile set are saved on the kernel stack, and the vector registers are saved if necessary. If the thread's process is inactive and the thread does not own any mutexes, it is inserted in the process ready queue. Otherwise, the thread is placed in the dispatcher ready queue specified by its priority. If the thread's execution was preempted, then it is placed at the front of the dispatcher ready queue. Otherwise it is placed at the end of the dispatcher ready queue.

A swap of the privileged context is performed (see Section 4.9.2.4), the dispatcher database lock is released, scalar registers are restored from the new thread's kernel stack, and an REI is executed to resume execution of the new thread.

4.9.3.6 Fork Queue Performance Data

Data is collected so that the distribution and average depth of the fork queue can be computed for each of the processors in a configuration. The data is collected by sampling the fork queue depth at each interval clock interrupt. The sampled depth is then used to select the element to increment in an array of counts. This data can be read via a performance data collection procedure; see Section 4.16.

4.9.4 Dispatcher Performance Data

One of the important parameters in cache and translation buffer design is the number of instructions that are executed between context switches. This is referred to as "headway," and together with locality of reference, determines the efficiency of a particular cache or translation buffer design.

In order to provide data for future system design and to provide a way in which alternative scheduling policies can be evaluated, the kernel will collect data on headway.

Headway is defined as the number of cycles that occur between context switches of threads. The data is collected in the form of a histogram of counts. That is, the number of cycles is used to index an array of counts and the element in the array incremented. Data is collected into two histograms: one for voluntary wait and one for preemption. This data can be read via a performance data collection procedure; see Section 4.16.

4.10 AST Processing

The kernel provides services for queuing and delivering asynchronous system traps (ASTs) to target threads. A combination of software state and hardware registers are used to determine the correct point to interrupt thread execution.

The PRISM architecture supplies three registers to control ASTs. Two of these registers (ASTEN, ASTSR) are part of the privileged context of a thread and are saved and restored at each context switch. The registers are:

- AST enable register (ASTEN)

The AST enable register records the AST Enable bits for kernel and user mode. A 1 bit signifies that ASTs are enabled for the respective mode, and a 0 bit signifies that ASTs are disabled. The Enable bit for the current mode can be changed with the Swap AST Enable instruction. It may also be changed in the privileged context block when a thread is not active (Terminated, Ready, Standby, or Waiting state).
AST request register (ASTRR)

The AST request register is used to request an AST for kernel or user mode. Writing this register sets the kernel or user AST Pending bit in the AST summary register.

AST summary register (ASTSR)

The AST summary register records the modes for which ASTs are pending. It is written when the AST request register is written, and may also be written in the privileged context block when a thread is not active.

Hardware constantly monitors the state of PS<iPL>, ASTEN<1:0>, and ASTSR<1:0>. Any time that PS<iPL> is zero, and an AST is both enabled and pending for the current or a more privileged mode, an AST delivery interrupt is generated at IPL 1.

In addition to the hardware registers, several items of software state relevant to AST processing are stored in the thread object:

- AST queue listheads

The thread object contains an AST queue for each of the modes kernel and user. AST objects that are to be delivered to kernel mode are inserted in the kernel AST queue and AST objects that are to be delivered to user mode are inserted in the user AST queue. AST objects of type normal are inserted at the tail of their respective queue, whereas AST objects of type special are inserted at the front of the kernel AST queue. Entries are always removed from the front of both of the queues.

- AST In Progress bits

An AST In Progress bit is maintained in the thread object for each of the modes kernel and user. These bits signify the modes for which normal ASTs are currently being executed and prevent further normal ASTs from being delivered to the respective modes. The bit is cleared when the AST procedure completes its execution.

- Acquired mutex list

A list of the mutexes that are currently owned by a thread is maintained in the thread object. Normal ASTs that execute in kernel mode are prevented from being delivered when the thread's acquired mutex list is not empty.

4.10.1 Queuing An AST Object

An AST object can be inserted in a target thread's AST queue by calling the kernel procedure k$insert_ast_queue. When an entry is inserted in a target thread's AST queue, the IPL is raised to 2 and the dispatcher database lock is acquired. The specified AST object is then inserted at either the head or tail of the appropriate AST queue. Subsequent processing is dependent on the current thread state.

There are three cases:

1. Thread state is Terminated, Ready, or Standby

   The appropriate AST Pending bit is set in the saved ASTSR register in the thread's hardware privileged context block. When the thread is dispatched for execution and the proper AST enabling conditions are present, an AST delivery interrupt occurs.

2. Thread state is Waiting

   The appropriate AST Pending bit is set in the saved ASTSR register in the thread's hardware privileged context block. In addition, the thread may be unwaited so that the AST can be immediately delivered. This is controlled by the mode parameter that was specified when the thread entered the Wait state. The actual Wait always occurs in kernel mode, and if a mode parameter specifying user is supplied, executive code must be ready to unwind to user mode to deliver an AST. An AST is delivered to a waiting thread by setting the Wait completion status in the thread object, and unwaiting the thread. Unwaiting a thread returns it to the Ready state.
ASTs directed to kernel and user mode are processed differently for threads in the Wait state, since the Wait status must be set differently.

If the target mode of the AST is kernel, certain conditions must be met before the thread is removed from the Wait state. If the AST is a special AST, the thread's saved ASTEN<0> is set, and the wait IPL is 0, the thread is removed from the Wait state with a Wait completion status of "kernel AST". If the AST is a normal AST, the thread's saved ASTEN<0> is set, the wait IPL is 0, the thread does not own any mutexes, and a normal AST is not active for kernel mode, then the thread is also removed from the Wait state with a Wait completion status of "kernel AST". When the thread is dispatched for execution, an AST delivery interrupt occurs, and the AST routine is executed in kernel mode. At the completion of the AST routine, the execution of the Wait is continued. The kernel Wait routine tests the Wait completion status and discovers that the Wait is to be repeated.

If the target mode of the AST is user, the wait mode is user, ASTEN<1> is set, and an AST is not active for user mode, then the thread is removed from the wait state with a status of "user AST." When the thread is dispatched for execution, it continues in the kernel Wait code, which tests the Wait status and finds that an AST is to be delivered to user mode. The condition "deliver user AST" is raised so that executive routines that are active have a chance to clean up data structures, and so forth. All intervening level routines clean up their structures and resignal. The highest-level handler, which is declared by the system service entry procedure, cleans up its structures and unblocks its caller with a status of "repeat" or "advance." Its caller is the Fault On Execute handler, which then examines the service status and finds it to be "repeat" or "advance."

The original argument registers are restored, the linkage register is restored, and a test is made to determine if the service is to be repeated or if it is to be continued in a second service whose vector immediately follows the specified vector in the entry page. The return PC (address in entry page) is advanced, if appropriate, and an REI is executed, whereupon an AST delivery interrupt occurs. The AST is delivered to user mode, the AST routine executes, then executes an AST Exit system service. When execution of the thread is continued, either the new or same system service is executed.

\ There should never be a case where the Wait is executed when the wait mode is user and the thread owns one or mutexes, or the wait mode is user and a normal AST is active in kernel mode. There is an implicit assumption that user-mode ASTs are never enabled when kernel-mode ASTs are disabled. If this were allowed, then a user AST could interrupt kernel mode when ASTs are disabled in kernel mode. \n
3. Thread state is Running

If the thread is running on the current processor, ASTRR is written specifying the mode that is to receive the AST. If, however, the thread is running on another processor, then the other processor must be directed to write its ASTRR register. The target processor is notified by raising the IPL to 7, acquiring the target processor's request summary lock, setting the AST request member in the request summary, storing the AST mode in the request value, requesting an interrupt on the target processor by writing the IPIR register with the target processor's number, and then releasing the processor's request summary lock and dropping the IPL back to 2. An interprocessor interrupt will occur on the target processor, causing it to write its ASTRR register with the specified Mode value. When proper enabling conditions are present, the target thread receives an AST delivery interrupt.

4–52  The Kernel
4.10.2 AST Delivery Interrupt

PRISM hardware constantly monitors the state of PS<CM>, PS<IPL>, ASTEN<1:0>, and ASTSR<1:0>. Whenever PS<IPL> equals 0 and an AST is enabled and pending for a mode that is equal or more privileged than the current mode, an AST delivery interrupt is generated at IPL 1. The correct Pending bit is automatically cleared in ASTSR by hardware.

When an AST delivery interrupt occurs, there are three cases to be handled:

1. The first entry in the kernel AST queue is a special AST.
2. The first entry in the kernel AST queue is a normal AST.
3. The previous mode is user, the kernel AST queue is empty, and the user AST queue contains one or more normal ASTs.

In response to the AST delivery interrupt, the volatile registers are saved on the kernel stack, the IPL is raised to 2, and the dispatcher database lock is acquired. The particular case to be handled is then determined by testing whether the kernel AST queue contains any entries, and if it does, whether the first entry in the queue is a special or normal AST. The fact that an AST delivery interrupt has occurred does not necessarily mean that an AST can actually be delivered. Certain enabling conditions must also be present; for example, a normal AST cannot interrupt a normal AST for the same mode.

The three cases are handled as follows:

1. Special AST is the first entry in the kernel AST queue.

   Special ASTs are delivered in kernel mode at IPL 1. Special ASTs interrupt normal ASTs for kernel mode, but cannot interrupt other special ASTs. The special AST object is removed from the kernel AST queue, the dispatcher database lock is released, and the IPL is lowered to 1. The routine specified by the kernel_routine parameter in the AST object is then called, specifying the AST object itself as an argument. When the specified routine returns, the IPL is raised to 2, the dispatcher database lock is reacquired, and the test to determine what case to handle is repeated. Executive code that must synchronize its execution with a special AST routine can do so by either raising IPL to 1 or by clearing ASTEN<0>.

2. Normal AST is the first entry in the kernel AST queue.

   Normal ASTs for kernel mode are delivered at IPL 0. Normal ASTs for kernel mode interrupt normal ASTs for user mode, but do not interrupt other normal ASTs for kernel mode. Proper enabling conditions must be present before the AST will actually be delivered. If there is currently no normal AST active for kernel mode and the current thread does not own any mutexes, the AST object is removed from the kernel AST queue, the AST Active bit for kernel mode is set in the thread object, the dispatcher database lock is released, and the IPL is lowered to 1.

   The procedure that was specified by kernel_routine parameter when the AST object was initialized is called with the address of the AST object as a parameter. When the specified procedure returns, the IPL is lowered to 0 and the procedure specified by the normal_routine parameter from the AST object is called with the arguments specified by the context and system_value parameters. When this routine returns, the IPL is raised to 2, the dispatcher database lock is reacquired, and the kernel-mode AST Active bit is cleared in the thread object. The test to determine which case to handle is repeated.

3. The previous mode is user, the kernel AST queue is empty, and the user AST queue contains one or more normal ASTs.

   Normal ASTs for user mode are delivered at IPL 0. Normal ASTs for user mode do not interrupt other normal ASTs for user mode. Proper enabling conditions must be present before the AST will actually be delivered. If there is currently no normal AST active for user mode and ASTEN<1> is set, the AST object is removed from the user AST queue, the AST Active bit for user mode is set in the thread object, the dispatcher database lock is released, and the IPL is lowered to 1.

The Kernel 4–53
The procedure that was specified by kernel_routine parameter when the AST object was initialized is called with the address of the AST object as a parameter. When the specified procedure returns, all state information that has been saved on the kernel stack is transferred to the user stack. A PC and PS are then pushed on the kernel stack which will transfer control to a sequence of code running in user mode at IPL 0. This code sequence calls the procedure specified by the normal_routine parameter from the AST object with the arguments specified by the context and system_value parameters. When this routine returns, an AST Exit system service is executed, the contents of the volatile registers are restored, and an REI is executed to resume thread execution.

The AST Exit system service calls the kernel procedure k$exit_ast, which raises IPL to 2, acquires the dispatcher database lock, and clears the AST Active bit for user mode in the thread object. If the user AST queue contains another entry, then another AST delivery interrupt is requested by writing the ASTRR register. The dispatcher database lock is released and the IPL is lowered to 0.

4.11 Interprocessor Interrupt

Interprocessor interrupts are used to notify target processors of pending work they are to perform. Before any request information can be established in the target processor's request summary, the requesting processor's IPL must be raised to 7, and the spin lock related to the target processor's request summary must be acquired. Appropriate members can then be set in the request summary, any other related information stored, and an interprocessor interrupt request sent to the target processor.

The request summary has the following type declaration:

```
TYPE
  k$request_summary : RECORD
    rq$ast : bit;
    rq$translation_flush : bit;
    rq$dispatch : bit;
    rq$instruction_flush : bit;
    rq$translation_flush_one : bit;
    rq$bug_check : bit;
    rq$freeze : bit;
    rq$frozen : bit;
    rq$unfreeze : bit;
END RECORD;
```

The fields in the request summary are used as follows:

- `rq$ast`: Request an AST by executing an MTPR specifying ASTRR and the associated request value.
- `rq$translation_flush`: Flush the entire translation buffer by executing a TBFLUSH instruction.
- `rq$dispatch`: Request dispatching by executing an MTPR instruction specifying SIRR and an IPL of 2.
- `rq$instruction_flush`: Flush the instruction buffer by executing an IBFLUSH instruction.
- `rq$translation_flush_one`: Flush a single translation buffer entry by executing an MTPR specifying TBIS, the virtual address stored in the request value, and the current address space number (ASN).
- `rq$bug_check`: Initiate an orderly crash of the system.
- `rq$freeze`: Freeze processor activity so that a system debugger can examine its state.
- `rq$frozen`: A status bit indicating that the processor has frozen its activity and stored its state in the appropriate restart parameter block (RPB) slot.
- `rq$unfreeze`: Unfreeze processor activity and continue execution.
When the target processor receives the interprocessor interrupt, it acquires the request summary spin lock and reads the request summary. If $rq$s$freeze$ is not set, the request summary and request value are saved, the request summary is cleared, and the spin lock is released. The saved request summary is then scanned and the indicated actions are performed.

If $rq$s$freeze$ is set, then a system debugger is active and the target processor is being requested to freeze its activity until explicitly told to continue. The processor stores its scalar register and internal processor registers in the appropriate RPB slot, clears $rq$s$freeze$, and sets $rq$s$frozen$ in the request summary. It then drops into a loop where it waits until the $rq$s$unfreeze$ member is set in the request summary. The system debugger executes, clears $rq$s$frozen$, and sets $rq$s$unfreeze$ in the request summary when it wants processor execution to continue. The processor discovers $rq$s$unfreeze$ is set, flushes its instruction buffer, clears $rq$s$unfreeze$, reloads its scalar and internal processor registers from its RPB slot, and repeats the test for actions to perform based on the request summary.

The kernel itself is the source of most interprocessor interrupt requests; however, memory management and debugger software outside the kernel have a need to request various actions.

The execution of all other processors in the system, excluding the current processor, can be frozen by calling a kernel function that has the following declaration:

```
PROCEDURE
  k$freeze ( )
  ) RETURNS k$sipl;
```

The IPL is raised to 7, the execution of all other processors in the system is frozen, and the previous IPL is returned as the function value. This routine does not return control to the caller until the execution of all other processors has been frozen and their state is stored in the respective RPB slots.

This routine is intended for use by system debuggers and should be called whenever the debugger is entered so that a consistent picture of the system can be examined.

The execution of all other processors in the system, excluding the current processor, can be resumed by calling a kernel procedure that has the following declaration:

```
PROCEDURE
  k$unfreeze ( IN ipl : k$sipl; )
```

Parameters:

ipl The level to which the IPL is to be lowered.

The execution of all other processors in the system is unfrozen and the IPL is lowered to the specified value. The instruction buffer is flushed on each of the processors including the current processor.

This routine is also intended for use by system debuggers and should be called when a debugging session has ended.

The translation buffer can be flushed on a target processor by calling a kernel procedure that has the following declaration:

```
PROCEDURE
  k$translation_flush ( IN processor : k$processor_number; )
```

Parameters:

processor The number of the target processor on which the translation buffer should be flushed.
The instruction buffer can be flushed on all processors, or only the set of processors that are currently executing threads belonging to the current thread's process, by calling a kernel procedure that has the following declaration:

```
PROCEDURE
  k$instructi on_flush (    
    IN all_processors : boolean;
  );
```

Parameters:

`all_processors` A boolean value that determines which instruction buffer(s) should be flushed.

If the value of the `all_processors` parameter is TRUE, the instruction buffer of all processors in the multiprocessor configuration is flushed. If the value of the `all_processors` parameter is FALSE, then the instruction buffer is flushed on only those processors that are executing threads belonging to the current thread's process.

A single entry can be flushed in the translation buffer of all processors, or only that set of processors that are currently executing threads belonging to the current thread's process, by calling a kernel procedure that has the following declaration:

```
PROCEDURE
  k$translation_flush_single (    
    IN virtual : POINTER anytype;
    IN all_processors : boolean;
  );
```

Parameters:

`virtual` The virtual address of any location in the page that is to be removed from the translation buffer.

`all_processors` A boolean value that determines which translation buffer(s) should be flushed.

If the value of the `all_processors` parameter is TRUE, the address specified by the `virtual` parameter is flushed from the translation buffer of all processors in the multiprocessor configuration. If the value of the `all_processors` parameter is FALSE, then the address specified by the `virtual` parameter is flushed from the translation buffer on only those processors that are executing threads belonging to the current thread's process.

### 4.11.1 Interprocessor Interrupt Performance Data

The number of interprocessor interrupts is maintained for each of the different types of requests that can be posted in the processor request summary with the exception of bug check. At each interprocessor interrupt, the type of request is determined and the appropriate counter is incremented. This data can be read via a performance data collection procedure (see Section 4.16).

### 4.12 Interval Timer Interrupt

The interval timer is used by the kernel for maintaining the system time, accumulating performance information, updating thread quantum, and timer queue maintenance.

There is an interval timer on each processor in a multiprocessor configuration and the interval timer causes an interrupt every millisecond when enabled.
4.12.1 System Time

The system time is maintained as a quadword count of 100ns intervals and is initialized to zero when the system is booted. Thereafter, the system time is updated at every interval timer interrupt by adding 10,000, which represents the number of 100ns intervals in a millisecond. It is the responsibility of executive software to establish the correspondence between system time and external time as it appears to users of the system.

\ When the executive establishes such a correspondence, it simply reads the current system time by calling a kernel function and then stores the returned system time and the externally specified time as a pair. Whenever the current time must be calculated, executive software again reads the system time, subtracts it from the stored system time, converts to the corresponding external time units, and then adds the external base time. \n
A single copy of the system time is simultaneously maintained by all processors in a multiprocessor configuration. This ensures that, should a processor fail, the time will continue to increase appropriately.

The format for system time is shown in Figure 4-3.

Figure 4-3: System Time

<table>
<thead>
<tr>
<th>31</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>BITS &lt;31:30&gt; OF SYSTEM TIME</td>
<td></td>
</tr>
<tr>
<td>BITS &lt;63:31&gt; OF SYSTEM TIME</td>
<td></td>
</tr>
</tbody>
</table>

The system time may be read by calling a kernel function that has the following declaration:

PROCEDURE
k$read_system_time ( ) RETURNS large_integer;

The current system time is read and returned as the function value.

4.12.2 Timer Queue Maintenance

The kernel maintains the time at which the first entry in the timer queue is to expire. Therefore, when an interval timer interrupt occurs the timer queue maintenance routine needs only to compare the current absolute time with the time at which the first entry in the timer queue is to expire. If the first entry has not expired, no other entry could have expired either. This comparison is always made after the absolute time has been updated.

\ In the interval timer interrupt service routine, the test is always made after the absolute time has been updated. In the kernel timer service routines, which run at IPL 2 and lock the dispatcher database, the test is always made after the time of first expiration is updated. This is necessary because there is no explicit synchronization between the two routines. If one of the routines makes a wrong decision while the other routine is updating either the absolute time or time of first expiration, then the other routine will make the right decision and the timer queue will get processed correctly. \n
If a timer has expired, the processor-specific timer queue fork block (TQFB) is inserted in the fork queue, and a software interrupt is requested at IPL 2. When the timer queue fork block (TQFB) is removed from the fork queue by the fork dispatcher, the timer queue is examined.
4.12.3 Quantum End

Threads are granted a quantum when they first begin execution. This may be when they first enter the balance set, or when they are granted a new quantum after their previous quantum has run out. Generally, quanta are used to control the maximum length of time a thread is allowed to execute at a given dispatch priority. At each interval timer interrupt, the quantum of the current thread is decremented. If the resultant value is less than or equal to zero, quantum end is said to have occurred, and the dispatcher must be invoked to decide what should be done with the thread’s priority and whether the thread’s process should remain in the balance set.

The dispatcher is invoked by placing the processor-specific quantum end fork block (QEFB) in the fork queue of the current processor (see Section 4.12.2).

4.12.4 Interval Time Performance Data

The amount of time spent in each of the modes kernel and user is sampled by the interval timer interrupt service routine. At each interval timer interrupt, the previous processor mode is used to select a counter to increment for kernel or user mode. This data can be read via a performance data collection procedure; see Section 4.16.

An array of counters is maintained in the processor control block (PB) to record the depth of the fork queue at each interval timer interrupt. The current depth of the fork queue is used to select the element to increment in the array of counters. This array of counters records the distribution of fork depth and can be read via a performance data collection routine; see Section 4.16.

4.13 Address Space Number (ASN) Management

The kernel provides for complete management of the assignment of address space numbers (ASNs). ASNs are used to tag translation buffer entries and therefore avoid flushing at every context switch.

The PRISM architecture allows an implementation of one of two possibilities: 1) A 16-bit ASN, or 2) no support for ASNs. In either case, there are insufficient ASNs to suffice over the entire time that a system can be active between bootstraps.

Although several strategies for the use of ASNs are possible, the kernel only uses ASNs to avoid translation buffer flushes. As each thread is dispatched, a new ASN is assigned if the new thread’s page table base register (PTBR) is not equal to the previous thread’s PTBR. This is accomplished by simply incrementing the ASN sequence number that is stored in the processor control block (PB). The result is stored in the hardware privileged context block of the thread and becomes the ASN when a SWPCTX instruction is executed.

Each processor maintains its own ASN sequence number and increments it when a new ASN must be assigned. When the incremented value is equal to the number of ASNs implemented by the host processor, a translation buffer flush is executed and the ASN sequence number is reset to zero.

4.14 Powerfail Recovery

Powerfail recovery support is provided by the kernel via power-up request and status objects. In conjunction with raising IPL, these objects provide a driver thread with the capability to interrupt its execution and/or have a status variable set when a power recovery interrupt is received by PRISM hardware. Although intended primarily for use by driver threads, power up request objects can also be provided to user-mode programs by the executive.

In order to request that a status variable be set or an AST queued, a proper object data structure must be allocated and then initialized by the appropriate kernel procedure. The object structure is then inserted into either the power-up request queue or the power-up status queue.

When power failure occurs, PRISM hardware automatically saves all volatile machine state in memory. This includes internal processor registers, scalar registers, and vector registers. The actual power down sequence cannot be initiated unless the interrupt priority level (IPL) of the processor is below 7. Thus, power failure can be inhibited for short periods by raising IPL to 7.
When power is restored, PRISM hardware restores all volatile machine state that was saved during the power down sequence and then generates a power-up interrupt (providing that memory is battery backed up). The power-up interrupt is generated at IPL 7.

The power-up interrupt service routine acquires the power-up status queue spin lock and examines the power-up status queue. If the queue is not empty, each entry in the queue is removed and the specified status variable is set to TRUE. The power-up status queue spin lock is then released, the power recovery fork block (PRFB) is inserted in the processor’s fork queue, and a dispatch interrupt is requested.

When the processor IPL drops to 1 or 0 (that is, possibly nested interrupt service routines finish their execution), a dispatch interrupt occurs, which causes the fork dispatcher to be executed. The fork dispatcher removes the power recovery fork block (PRFB) from the fork queue and executes the action routine associated with this type of fork block. The power-up action routine acquires the power-up request spin lock and examines the power-up request queue. If the queue is not empty, each entry in the queue is removed and the associated AST is queued to the target thread. The power-up request spin lock is then released and fork processing is continued.

Both status queue and request queue entries are one-shot operations. Thus, once a power recovery interrupt has occurred, no further operations are performed, unless new requests are placed in the appropriate queues.

The above scenario guarantees that one and only one processor in a multiprocessor configuration will gain access to the corresponding spin locks and empty the request queues. In addition, all processors go through the sequence of acquiring the locks so that there is no race condition that would enable a thread to execute before its AST is queued or its status variable set.

### 4.14.1 Driver Power Failure Example

In general, a function processor thread requests that both an AST be delivered and a status variable be set when a power failure and subsequent recovery occur. The AST enables the driver to break out of the current execution path to requeue the power-up request and status objects and initialize the device and its units. The status variable allows the driver code to ensure that it can safely load device registers.

The following program fragment illustrates the proper way to load device registers to ensure that a power failure does not corrupt the registers in a manner that would cause an unpredictable I/O transfer:

```plaintext
VARIABLE
  csr_address : POINTER integer;
  interrupt_object : POINTER k$control_object; !(k$c_interrupt);
  power_status_object : POINTER k$control_object; !(k$c_power_status);
  status_value : boolean;

BEGIN
  ! Allocate, initialize, and connect an interrupt object.
  !
  interrupt_object = ALLOCATE(k$control_object(k$c_interrupt));
  k$initialize_interrupt(interrupt = interrupt_object,
    service_routine = driver_interrupt_routine,
    context = csr_address,
    offset = "100"x,
    ipl_level = 5);
  k$connect_interrupt(interrupt = interrupt_object);

The Kernel 4-59
! Allocate, initialize, and insert a power-up status object.
!

power_status_object = ALLOCATE(k$control_object(k$sc_power_status));
k$initialize_power_status(power_status = power_status_object);
k$insert_power_status_queue(power_status = power_status_object,
                                 state = status_value);

.
.

IF k$synchronize_execution(interrupt = interrupt_object,
                           synch_routine = register_load,
                           context = csr_address) THEN

  ! Power has failed. Perform reinitialization etc.
  !
  .
  .

END IF;

PROCEDURE register_load (context)
  OF TYPE k$synchronized_routine;

VARIABLE
  previous_ipl : k$ipl;

BEGIN

  ! Prevent a power down sequence by raising IPL to 7.
  !
  previous_ipl = k$set_interrupt_priority_level(ipl = 7);

  ! Check to determine if power failed. If the power has not failed,
  ! then it is safe to load device registers. Otherwise just return.
  !
  IF NOT status_value THEN
    ! Load device registers.
    !
  END IF;

  ! Restore interrupt priority and return state of status variable
  ! as the function value.
  !
  previous_ipl = k$set_interrupt_priority_level(ipl = previous_ipl);
  RETURN(status_value);
END register_load;
4.15 Vector Enable Fault

A vector enable fault is generated whenever an attempt is made to execute a vector instruction with the Vector Enable bit clear in the current processor status (PS<VEN> = 0). This capability is used to detect the first attempt by a thread to use the vector registers and to optimize the saving and restoring of vector registers on context switches.

When a vector enable fault is generated, exception dispatching software gains control and checks to determine if the thread has a vector register save area allocated.

If a vector register save area is allocated, then the current processor has a vector unit (the affinity cannot be changed to a processor without a vector unit once a vector register save area has been allocated) and the current thread has previously established the contents of the vector registers. A PC and PS pair are pushed on the stack with PS<VEN> = 1, and PS<IPL> = 2. An REI instruction is then executed to enable vector instructions, prevent context switching by raising IPL, and continue execution in the fault routine. If the thread was the last thread to use the vector registers on the current processor, then the vector registers need not be loaded. Otherwise the vector registers are loaded from the vector register save area and the address of the current thread is stored in the processor block as the last user of the vector registers. PS<VEN> is set is the fault PS and an REI is executed to continue thread execution.

If a vector register save area is not allocated, then a check is made to determine if the thread can execute on any of the processors in the configuration that have a vector unit. This check is performed by forming the logical product of the thread's affinity with the set of processors that have a vector unit. If the result is the null set, a reserved instruction exception is reflected to the processor mode in which the fault occurred. Vector instructions are not emulated.

If, however, there are one or more processors on which the thread can execute and which have vector units, then a Set Affinity kernel function is executed to move execution of the thread to a processor with a vector unit. Note that the current processor may have a vector unit, in which case no context switch will take place.

Executing now on a processor with a vector unit, a vector register save area is allocated and a PC and PS pair are pushed on the stack with PS<VEN> = 1 and PS<IPL> = 2. An REI instruction is then executed to enable vector instructions, prevent context switching by raising IPL, and continue execution is the fault routine. The address of the vector register save area is stored in the current thread object, the vector registers are cleared, and the address of the current thread is stored in the processor block as the last user of the vector registers. PS<VEN> is set is the fault PS and an REI is executed to continue thread execution.

4.16 Performance Data Collection

The kernel collects various categories of performance data during its execution so that both the designers and users of the system can analyze and improve its performance. The data structures required to record this data are private to the kernel and, therefore, are not directly accessible to executive software. Executive software can retrieve this data, however, by calling a kernel procedure that returns the desired category of data.

4.16.1 Computable and Waiting Threads

The number of currently computable and waiting threads can be obtained by calling a kernel procedure that has the following declaration:

```prologue
PROCEDURE
  k$get_thread_state_data (
    OUT spin_count : integer;
    OUT ready_state : integer;
    OUT wait_state : k$wait_reason_data;
  )
```

The Kernel 4–61
Parameters:

spin_count  A variable that receives the accumulated spin count of the dispatcher database lock.
ready_state  A variable that receives the number of threads that are currently in a Ready state.
wait_state  An array variable that receives the number of threads that are waiting for each of the possible reasons that are defined by the executive.

The current number of threads waiting for each of the possible reasons is maintained directly by the kernel Wait routines. The number of computable threads, however, must be derived by scanning the dispatcher ready queues. A simple count cannot be maintained because the dispatcher ready queues can contain threads whose processes have been removed from the balance set. The number of computable threads includes threads that are in the Standby state.

4.16.2 Thread Data

Performance data for a thread can be obtained by calling a kernel procedure that has the following declaration:

```plaintext
PROCEDURE k$get_thread_data ( 
  IN thread : POINTER $dispatcher_object; !(k$s_thread); 
  OUT kernel_ticks : integer; 
  OUT user_ticks : integer; 
  OUT preemption_switches : integer; 
  OUT voluntary_switches : integer; 
  OUT quantum_ends : integer; 
 );
```

Parameters:

thread  The virtual address of a dispatcher object of type thread.
kernel_ticks  A variable that receives the accumulated count of the number of clock ticks that occurred while the thread was in kernel mode.
user_ticks  A variable that receives the accumulated count of the number of clock ticks that occurred while the thread was in user mode.
preemption_switches  A variable that receives the number of times the thread has been context switched because it was preempted.
voluntary_switches  A variable that receives the number of times the thread has been context switched because it voluntarily entered a wait state.
quantum_ends  A variable that receives the number of times the thread has experienced quantum end.

Processor mode data is sampled at every interval timer interrupt and the corresponding kernel or user count is updated for the current thread. At each context switch away from a thread either the number of preemption or voluntary switches is incremented, depending on whether or not the thread gave up the processor voluntarily. The number of quantum ends is incremented each time the thread exceeds its time quantum.

4.16.3 Fork Queue Depth

The fork queue depth data can be obtained for a processor by calling a kernel procedure that has the following declaration:

```plaintext
PROCEDURE k$get_fork_queue_data ( 
  IN processor : k$processor_number; 
  OUT depth : k$fork_queue_depth_data; 
 );
```
Parameters:

processor  The number of the processor for which the data is to be collected.
depth  An array variable that receives the sampled fork queue depth data for the specified processor.

The fork queue depth data is computed by sampling the fork queue depth at each interval clock interrupt. Each entry in the depth array is a summation of the number of times that the fork queue was found to be at the particular depth. The first entry in the array records the number of times the depth was 0; the second, the number of times the depth was 1; and so on.

4.16.4 Context Switch Headway Data

Context switch headway data can be obtained by calling a kernel procedure that has the following declaration:

PROCEDURE k$get_headway_data (  
    OUT preemption : k$headway_data;  
    OUT voluntary : k$headway_data;  
);  

Parameters:

preemption  An array variable that receives the context switch headway histogram collected on context switches that were caused by thread preemption.
voluntary  An array variable that receives the context switch histogram collected on context switches that were caused by voluntary waits.

At each context switch the total number of cycles that have occurred since the last context switch is divided by 1024 (accomplished by shifting) to compute the index of the element in the headway array that should be incremented. This index value is then compared with a constant which is the extent of the headway data array. If the index is greater than the extent value, the last entry in the headway array is incremented. Otherwise, the element in the headway array addressed by the index value is incremented. The specific headway array that is referenced depends on whether the context switch occurred as the result of thread preemption or voluntary wait.

4.16.5 Interprocessor interrupts

The number of interprocessor interrupts for each of the various request types can be obtained for a processor by calling a kernel procedure that has the following declaration:

PROCEDURE k$get_processor_request_data (  
    IN processor : k$processor_number;  
    OUT spin_count : integer;  
    OUT asts : integer;  
    OUT translation_flushes : integer;  
    OUT dispatches : integer;  
    OUT freezes : integer;  
    OUT instruction_flushes : integer;  
    OUT translation_flush_singles : integer;  
);  

Parameters:
processor The number of the processor for which the data is to be collected.

spin_count A variable that receives the accumulated spin count of the request summary lock for the specified processor.

asts A variable that receives the number of AST requests that have occurred for the specified processor.

translation_flushes A variable that receives the number of translation buffer flushes that have occurred for the specified processor.

dispatches A variable that receives the number of dispatch requests that have occurred on the specified processor.

freezes A variable that receives the number of freeze execution requests that have occurred on the specified processor.

instruction_flushes A variable that receives the number of instruction buffer flushes that have occurred on the specified processor.

translation_flush singles A variable that receives the number of translation buffer single requests that have occurred on the specified processor.

The number of interprocessor interrupts for each of the request types is maintained in the processor control block (PB) and is incremented as each type of request is received.

4.16.6 Interrupt Object Data

The interrupt data for an interrupt vector can be obtained by calling a kernel function that has the following declaration:

```
PROCEDURE
  k$get_interrupt_data ( 
    IN offset : integer;
    OUT spin_count : integer;
    OUT count : integer;
  ) RETURNS boolean;
```

Parameters:

offset The offset into the system control block (SCB), in longwords, of the vector for which the interrupt data is to be collected.

spin_count A variable that receives the accumulated spin count of the interrupt object that is associated with the specified SCB vector.

count A variable that receives the number of interrupts that have occurred through the specified SCB vector.

If an interrupt object is not connected to the system control block (SCB) vector, a function value of FALSE is returned. Otherwise, the requested data is returned with a function value of TRUE.

4.16.7 Device Work Queue Data

The contention data for a device work queue can be obtained by calling a kernel procedure that has the following declaration:

```
PROCEDURE
  k$get_work_queue_data ( 
    IN work_queue : POINTER k$control_object(work_queue);
    OUT spin_count : integer;
    OUT depth : integer;
  );
```
Parameters:

work_queue  The virtual address of a control object of type work queue.
spin_count  A variable that receives the accumulated spin count of the lock for the specified work queue.
depth       A variable that receives the current number of entries in the work queue.

4.16.8 Processor Mode Data

The number of interval timer interrupts that have occurred while the previous mode was user and kernel can be obtained for a processor by calling a kernel procedure that has the following declaration:

PROCEDURE
  k$get_mode_data (  
    IN processor : k$processor_number;  
    OUT kernel_ticks : k$kernel_mode_data;  
    OUT user_ticks : integer;  
    OUT idle : large_integer;  
  );

Parameters:

processor  The number of the processor for which the data is to be collected.
kernel_ticks  A variable that receives the number of interval clock interrupts that have occurred on the specified processor when the previous mode was kernel.
user_ticks  A variable that receives the number of interval clock interrupts that have occurred on the specified processor when the previous mode was user.
idle  A variable that receives the number of cycles consumed by the idle thread on the specified processor.

Processor mode data is sampled at every interval timer interrupt and the corresponding kernel or user count is incremented. The cycle count of the idle thread is collected each time it is dispatched for execution.

4.16.9 Mutex Contention Data

The total number of times a mutex has been acquired and the number of times a thread had to actually wait before acquiring the mutex can be obtained by calling a procedure that has the following declaration:

PROCEDURE
  k$get_mutex_data (  
    IN mutex : POINTER k$dispatcher_object; !k$mutex);  
  OUT acquires : integer;  
  OUT waits : integer;  
  );

Parameters:

mutex  The virtual address of a dispatcher object of type mutex.
acquires  A variable that receives the total number of times the mutex has been acquired.
wants  A variable that receives the number of times a thread actually had to wait before being able to acquire the mutex.
4.16.10  Dispatcher Object Wait Queue Depth Data

The current depth of the wait queue for a dispatcher object can be obtained by calling a procedure that has the following declaration:

```
PROCEDURE
  k$get_wait_queue_data (
    IN object : POINTER k$dispatcher_object;
    OUT depth : integer;
  )
```

Parameters:

- `object`  
  The virtual address of a dispatcher object.

- `depth`  
  A variable that receives the current depth of the dispatcher object's wait queue.
CHAPTER 5
OBJECT ARCHITECTURE

5.1 Overview

5.1.1 Introduction
This chapter describes the software architecture of objects. It describes what objects are, and defines the data structures and operations necessary to support objects.

5.1.2 What is an Object?
Objects are abstract elements provided by an operating system that may be accessed by a user or a program. Typically, objects are defined in terms of the operations that may be performed upon them (for example, create, clear, set, get information, wait, delete) and their relationships to other objects. The reason for categorizing these elements as objects is to provide a single, standardized set of rules for creating, naming, protecting, accessing, and managing them. For example, each object has a unique ID value (called an object ID) which may be used to identify it. Objects at the job and process levels are only directly expressible by threads in that job or process.

5.1.3 Scope
It is important to understand that this chapter only defines the architecture of objects, not all object types. It is necessary that some objects or parts of objects be defined as part of this architecture.

5.1.4 Requirements and Goals

- Software development goals
  - Provide an extensible, yet rigorous framework for the definition and manipulation of executive-controlled data structures.
  - Maintain management consistency. The management of objects, in terms of actions taken to fulfill service requests, should be as object-type independent as possible. For example, standard routines and procedures can be established for determining whether access to an object should be granted.
  - Provide new object definition support. It should be possible to add new object types to the system without having to modify existing system code. This means that the interface between the kernel/executive system software and objects must be well-defined, and that the kernel/executive need not have knowledge of the internals of all objects.

- Interface goals
  - Provide consistent specification. The ways in which each object in the system may be specified by users should be minimized and kept consistent with the manner in which other objects are referenced.
Provide consistent operations. There are some operations that apply to a set of objects within the system, such as wait. The definition of what these operations mean to each object should be kept simple and similar to their definition for other objects.

Support level independence. Where possible, the operations that may be performed on an object type, and the behavior of that object, should not be dependent upon the level (system, job, or process) at which that object has been created. This allows applications to be developed in the relative safety of process and job levels before being moved to a more shareable level, with minimal change in behavior.

Provide security and protection. The method of determining which objects a user may refer to, and which operations may be performed on those objects, should be the same for all objects. This is the basis for Mica security.

5.1.5 Functional Description

The object architecture runs in kernel mode at IPL 0. Through the use of mutexes, object architecture procedures can simultaneously execute on multiple processors.

The object architecture provides a framework for creating object-specific services. These services include creating, deleting, allocating, referencing, name translating, and getting information about objects. For example, the object service to create an event, and the object service to create a thread both invoke the same object architecture-defined routine to create the object.

The object architecture provides a hierarchical visibility structure for objects. When an object is created, it is placed at one of three levels: system, job, or process. Objects at the system level are visible to all threads on the system. Objects at the job level for a particular job are only visible to threads in that job. Objects at the process level for a particular process are only visible to threads in that process. For example, a thread cannot access an object that is at the process level for another process, because it cannot express an object ID for that object.

Each level can contain one or more object containers to catalog objects at that level. There are two types of object containers at the process level: Process-private object containers and display object containers. Objects stored in a process-private object container are only visible to the process with which the container is associated. Objects stored in a display object container are visible to the associated process, and any of its descendant processes.

Objects are referred to by object ID. If a program refers to an object name, this name must be translated to an object ID. An object name is unique within a container for each object type at each processor mode. When a user attempts to refer to an object using an object ID, the user's access rights are compared to the access rights associated with an object. If there is a match, the user may access the object.

An object may be allocated to a user, resource ID, job, process, or thread. This allows objects to be shared among restrictive classes of users.

An object ID is deleted when the object container holding the corresponding object is deleted, or when the object ID is explicitly deleted. The object itself, however, is not deleted until the ID is deleted, and there are no outstanding references to the object.

Table 5-1 summarizes key object architecture terms and components.

<table>
<thead>
<tr>
<th>Table 5-1: Object Architecture: Terms and Definitions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Term</td>
</tr>
<tr>
<td>---------------------</td>
</tr>
<tr>
<td>Object ID</td>
</tr>
</tbody>
</table>

5-2 Object Architecture
### Object Architecture: Terms and Definitions

#### Object Identification and Names

<table>
<thead>
<tr>
<th>Term</th>
<th>Type</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Principal Object ID</td>
<td>Object ID</td>
<td>Associated with an object at object creation. An object has exactly one principal object ID.</td>
</tr>
<tr>
<td>Reference Object ID</td>
<td>Object ID</td>
<td>Optionally associated with an object. An object may have zero, one, or more reference IDs.</td>
</tr>
<tr>
<td>Object Name</td>
<td>Character String</td>
<td>Together with type and mode, an object name can be translated to an object ID. The combination of object type, mode, and name string is unique within a single object container.</td>
</tr>
<tr>
<td>Object Name Table</td>
<td>Data Structure</td>
<td>Tracks both logical names and object names within an object container. When an object container is created, a name table is also allocated, and that address is stored in the object container's body.</td>
</tr>
</tbody>
</table>

#### Object Hierarchy

<table>
<thead>
<tr>
<th>Term</th>
<th>Definition</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Object Container</td>
<td>Object Type</td>
<td>Objects of this type contain pointers to other objects. They are used to organize large numbers of objects.</td>
</tr>
<tr>
<td>Object Level</td>
<td>—</td>
<td>Indicates the scope of visibility of an object container.</td>
</tr>
<tr>
<td>System Level</td>
<td>Object Level</td>
<td>Objects at this level are potentially accessible to all processes on the system.</td>
</tr>
<tr>
<td>Job Level</td>
<td>Object Level</td>
<td>Objects at this level are potentially accessible to all processes in a given job.</td>
</tr>
<tr>
<td>Process Level</td>
<td>Object Level</td>
<td>Objects at this level are potentially accessible to all threads in a given process. Containers at this level can be either display or private.</td>
</tr>
<tr>
<td>Display Object Container</td>
<td>Object Container</td>
<td>Objects in such containers are accessible to a given process and all of its descendants.</td>
</tr>
<tr>
<td>Private Object Container</td>
<td>Object Container</td>
<td>Objects in such containers are accessible only to a given process, and not to its descendants.</td>
</tr>
<tr>
<td>Container Directory</td>
<td>Data Structure</td>
<td>Used to organize large numbers of object containers. All threads have the same system container directory. All threads in a job have the same job container directory. All threads in a process have the same process container directory.</td>
</tr>
<tr>
<td>Object Header</td>
<td>Data Structure</td>
<td>Fixed-format data structure that contains object type-independent data. This header is used by the executive without necessarily knowing the type of object it is accessing.</td>
</tr>
<tr>
<td>Object Body</td>
<td>Data Structure</td>
<td>A data structure that is specific to an object type.</td>
</tr>
</tbody>
</table>

Object Architecture 5-3
Table 5-1 (Cont.): Object Architecture: Terms and Definitions

<table>
<thead>
<tr>
<th>Term</th>
<th>Definition</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Object Type</td>
<td>–</td>
<td>Object type determines what operations can be performed on an object.</td>
</tr>
<tr>
<td>Object Type Descriptor (OTD)</td>
<td>Data Structure</td>
<td>Describes what operations are supported for what object types. There is one OTD for each object type.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Term</th>
<th>Definition</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Object Service Routines</td>
<td>System Routines</td>
<td>Implement operations that can be performed on objects. Some object service routines are particular to a certain type of object; others are supported across all object types.</td>
</tr>
<tr>
<td>Object Allocation Block</td>
<td>Data Structure</td>
<td>Contains information about object allocation.</td>
</tr>
</tbody>
</table>

5.1.6 Object-Related Operations

The following types of operations can be performed on most types of objects:

- Creating an object
- Protecting an object
- Translating an object ID
- Deleting an object
- Creating references to an object
- Making a temporary object
- Marking a new object as temporary
- Allocating an object
- Deallocating an object
- Getting information about an object
- Changing the name of an object

5.1.7 Summary of Proposed Changes to Object Architecture

We propose that the following changes be made to the existing object architecture chapter:

- Removal of all ULTRIX dependencies. These include clone procedures and executive actions.
- Addition of Pillar definition records.
- Deduction of quota rules.

5-4 Object Architecture
5.2 Functional Interface and Description

An object is a set of data structures representing an abstraction. Processes and events are examples of objects. In the case of an event object, the object represents the abstraction of an event that either has or has not taken place.

An object consists of a standard data structure called the object header, described in Section 5.4, and an object type-dependent data structure called the object body.

5.2.1 When Should an Element in the Executive be an Object?

An element should be an object when it:

- Needs a name for sharing.
- Needs to be referenced by user-mode software, and there are multiple instances of these elements.
- Needs to be protected via ACLs. Exceptions to this include structures that are exported by an RPC, such as job controller queues.
- Can be allocated.

No object can be dependent on another object such that there is an order dependence during object ID deletion. This allows various rundown operations to delete object IDs in any order, yet the rundown proceeds efficiently.

5.2.2 Object IDs

When an object is created, it is assigned a 64-bit value called its object ID. This value provides the fastest means for a user-mode routine to identify and access the object.

When an object is created, the ID returned is called the principal ID. Every object has only one principal ID. An object may also have one or more reference IDs which also point to the object. Reference IDs are discussed in Section 5.7.9.

The principal ID of an object that is accessible by two processes is the same in both processes. This extends to objects at all levels (including those in ancestors' process-display object containers, described in Section 5.5.9). This property allows IDs of shared objects to be communicated between cooperating threads.

5.2.3 Object ID Format

Object IDs are used to uniquely identify objects within the system. Object IDs are not addresses of objects, but values that may be used to generate the addresses of objects.

Object IDs are comprised of the following fields:

- Sequence number (\texttt{obj\_id$seq\_number\_low}, \texttt{obj\_id$seq\_number\_high})
- Level (\texttt{obj\_id$level})
- Object container index (\texttt{obj\_id$obj\_index})
- Container directory index (\texttt{obj\_id$con\_index})

The format of an object ID is shown in Figure 5-1.

Object ID value zero (0) is reserved for use as an invalid value.
5.2.4 Object ID Sequence Numbers

Three object ID fields are used to locate the address of the object. They are the level field, the object container index field, and the container directory index field. In addition, the sequence number high and low are used to help catch programming errors. Such an error might occur if a program used an object ID after the corresponding object had been deleted, and another object had been created with the same ID as the previously deleted object.

For example, consider a program with the following error. First, the program creates a FIZBOT object (which receives a corresponding object ID). Sometime later, the program deletes the FIZBOT object and creates a WALZOL object. Suppose that, due to a programming error, an operation on the FIZBOT object is attempted (such as a wait operation). If the WALZOL object had coincidentally received the same object ID that had been used for the FIZBOT object, a wait on the wrong object might go undetected.

To catch this type of error, each time an object ID is created, the sequence number fields (two 10 bit fields) are different each time an object ID is assigned to the same "slot".

When an object ID is created, the old sequence high field, which was saved, is incremented and reset to zero in the overflow case. In our example above, the improper use of the FIZBOT object ID would be detected and flagged as an error (no such object). Notice that the sequence high field cycles every 1024 allocations of an object ID.

The sequence low field receives a random value.

The use of sequence numbers does not totally eliminate the problem; it just reduces it to an extremely small probability for most programming situations.

5.2.5 Object Containers

Since programs are likely to have many objects at any point in time, it is desirable to be able to group objects together in some logical manner. For this purpose, a type of object called an object container is defined. When creating an object, it is necessary to specify the ID of the object container in which a pointer to the newly created object is to be stored.

From the executive's point of view, object containers are used when translating an object ID into a pointer to the object.

5.2.6 Object Levels

There are three levels of object visibility: process, job and system. The principal object ID level field determines the visibility of an object. The software process control block (SWPCB) contains level directors for the object levels. For each level, there is a pointer to the container directory corresponding to that level.

5–6 Object Architecture
5.2.6.1 System Level

The purpose of the system level is to provide a place to keep objects shared by multiple jobs. Implementation of the group concept supported by VMS uses this level of sharing, by creating an object container at the system level that is only accessible to threads with the proper group identifier.

Objects at this level are potentially accessible by all processes in the system. If access to an object at this level is to be restricted, then access protection must be explicitly assigned.

5.2.6.2 System Level Director Mutex

A system-wide mutex exists for synchronizing access to the system level containers. A pointer to this mutex is located in each software process control block. This mutex is acquired for object ID translations at the system level and for other system-level operations, such as Delete Object ID.

5.2.6.3 Job Level

The purpose of the job level is to allow objects to be shared by all processes within a single job. If access to an object at this level is to be restricted so that only some processes within a job may access it, then access protection must be explicitly assigned.

5.2.6.4 Job Level Director Mutex

A job-wide mutex exists for synchronizing access to the job level containers. A pointer to this mutex is located in each process control block. This mutex is acquired for object ID translations at the job level, and for other job level operations.

5.2.6.5 Process Level

The process level has two types of containers, display and private.

The purpose of process-private object containers is to provide a location for objects that are not accessible to any other process. No access protection needs to be assigned to objects in this type of object container. This simplifies the programming effort, and also provides the fastest possible object access.

The purpose of process-display object containers is to provide a location for objects that are accessible to the associated process, and to all descendant processes of that process. Objects in this object container do not typically require any access protection. Note, however, that if access protection is assigned to objects at the process level, it is checked at the time of each access.

It is important to note that the process-display object containers are only accessible to a process and its descendant processes.

5.2.6.6 Process Level Director Mutex

A job-wide mutex exists for synchronizing access to the process level containers. A pointer to this mutex is located in each process control block. This mutex is acquired for object ID translations at the process level, and for other process-level operations.

This mutex must be job-wide to allow proper synchronization on display containers which are shared among a process and its descendants.

5.2.7 Container Directory Index Field of the Object ID

The container directory index field is used as an index into the object array of the container directory. This yields a pointer to the object container, which contains a pointer to the object represented by the object ID.
5.2.8 Container Directory

For each level there is a corresponding container directory. All threads share the same system container directory. All threads within the same job share the same job container directory. All threads within a process shares the same process container directory.

The total number of container directories within the system at any given time is equal to the number of jobs plus the number of processes plus one (for the system level container directory).

The container directory provides a structure containing pointers to all the object containers within that level. It also provides a method of naming object containers.

5.2.8.1 Object Index Field of the Object ID

The object index field is used as an index into the object array of the object container located by the directory index. This yields a pointer to the object header of the object represented by the object ID.

5.2.9 Expressibility of Object IDs

One characteristic of the architecture is that threads are not able to directly express the ID of all objects. Here, the term "directly express" means to generate an ID value that corresponds to an object.

All threads are able to directly express the ID of all objects at the system level. However, objects at the job and process levels are only directly expressible by threads in that job or process. For example, there is no object ID value a thread in job X can use to directly refer to an object at the job or process level in job Y. Also, a thread in process Q cannot directly refer to process level objects in any other process, unless the object is in a process-display object container of an ancestor process (process-display object containers are discussed in Section 5.2.6.5).

5.2.10 Shareability of Object IDs

If two threads can both express the principal ID of an object, then those ID values are the same value. This allows cooperating threads in different processes or jobs to communicate object IDs to one another.

Notice that this property even extends to objects in ancestor process-display object containers. This allows a thread in one process to communicate an object ID of an object in its process-display object container to a thread in a descendant process (process-display object containers are discussed in Section 5.2.6.5).

5.2.11 Translation of an Object ID to an Object Header Address

In order to translate an object ID to the address of the associated object, Mica performs the following steps:

1. Uses the level number as an index into the process control block to locate the corresponding level directory mutex.
2. Acquires the directory mutex.
3. Uses the level number as an index into the process control block to locate the corresponding container directory.
4. Compares the container directory index field to the size of the table to ensure that index is within the object array.
5. Uses the container directory index field to index into the object array.
6. Checks the object array element to ensure it contains the address of an object header.
7. Uses the resulting system address to locate the object container header for the object container.

5-8 Object Architecture
8. Compares the object index field to the size of the table to ensure that index is within the object array.

9. Uses the object index field to index into the object array. Checks the resulting value to ensure that it is the address of an object.

10. Compares both sequence number fields in the object header to the sequence number fields in the object ID. To do this, Mica checks the address of the object header, which is found in the object array. If they are not identical, then that object ID is not valid.

11. Increments the pointer count field in the object header so the object's storage cannot be deallocated while a pointer to the object is held.

12. Releases the directory mutex, and returns a pointer to the object body to the caller.

The routine obj$reference_object_by_id described in Section 5.7.6 provides the mechanism to translate an object ID to the address of the object header. All routines within the executive use obj$reference_object_by_id for translating object IDs.

Figure 5–2 indicates how the structures fit together.

5.2.12 Container Directory IDs

It is necessary for programs to find container directories for each level. Since no consumer of the object architecture may have any knowledge about the construction of an object ID, programs call a system service that returns the object ID of the desired container directory.

The container directories are named as follows:

- The system level container directory is named exec$system_container_directory.
- The job level container directory is named exec$job_container_directory.
- The process level container directory is named exec$process_container_directory.

The exec$translate_object_name service may be used to obtain the object ID of a container directory by translating one of these three names. The caller must specify the container directory name for both the object name to locate and the container directory in which to locate the name.

5.2.13 Object Access By ID Only

One guideline to follow in system software design is that user-mode access to objects is only allowed by object ID, and is not allowed by name. If the object must be located via a name, the translation from name to ID is performed in a separate call preceding the object access call. This simplifies the coding of the object service routines, since they do not have to do any name translations or deal with logical names.

5.2.14 Object Service Routines

Each type of object has an associated set of services called object service routines. The service routines for a particular object type perform the operations supported by that object type.

For example, a FRAMITZ object type may define the operations:

- create_framitz
- get_framitz_information
- clear_framitz
- juggle_framitz
- read_framitz
- close_framitz

Each of these operations is represented by a separate object service routine. These object service routines can be added to the system dynamically, but only as a complete group.
Every set of object service routines must supply a method for creating the object and providing information about the object. These services are named create_object_type, and get_object_type_information. Each set of object service routines must also implement routines to provide system defined object type-specific operations.

These operations are:
- Remove
- Delete
- Shutdown
- Allocate
- Deallocate

These operations are used by the executive, and, in some cases, by the object's service routines. The availability and location of these functions, as well as some status and control information, are provided in object type descriptors (OTDs).

5.3 Object Type Descriptor (OTD)

There is a single object type descriptor (OTD) object for each object type defined by the system. All OTD objects are created in a single system container.

An OTD object, hereafter referred to as just OTD, contains general information about an object type, not a particular instance of an object type. For example, an OTD indicates whether an object type supports the notion of wait. Anything an OTD specifies about an object type applies to all instances of that object type.

Part of the information in an OTD is the location of a number of routines that may be called by the executive. These routines have standard definitions across all object types but require object type-specific processing. These routines are located by the executive via standard offsets into the OTD. The operations are listed in the previous section.

Users of an object type do not need to know anything about the routines pointed to by the OTD. The designer of an object type, however, must specify and implement them.

5.3.1 Object Type Descriptor Body Format

```
e$object_type_descriptor : RECORD
  otd$type : e$object_type;
  otd$objhdr_listhead : e$linked_list;
  otd$count : integer;
  otd$dispatcher_object_offset : integer;
  otd$access_mask : POINTER e$access_mask;
  otd$allocation_listhead_offset : integer;
  otd$create_disable : bit; ! 0 - enable, 1 - disable
  otd$mutex : k$dispatcher_object (mutex);
  otd$allocate : obj$allocate
  otd$deallocate : obj$deallocate
  otd$remove : obj$remove
  otd$delete : obj$delete
  otd$shutdown : obj$shutdown
END RECORD;
```

All fields of the OTD have a standard definition. This allows the executive to use the information in an OTD without having detailed knowledge of the corresponding object type.

Object Architecture 5–11
Figure 5–3: Object Type Descriptor (OTD) Format

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>OTD$TYPE (Type)</td>
<td></td>
</tr>
<tr>
<td>OTD$OBJHDR_LISTHEAD (Linked List)</td>
<td></td>
</tr>
<tr>
<td>OTD$COUNT (Count)</td>
<td></td>
</tr>
<tr>
<td>OTD$DISPATCHER_OBJECT_OFFSET (Dispatcher Offset)</td>
<td></td>
</tr>
<tr>
<td>OTD$ACCESS_MASK (Access Mask)</td>
<td></td>
</tr>
<tr>
<td>OTD$ALLOCATION_LISTHEAD_OFFSET (Allocation Listhead Offset)</td>
<td></td>
</tr>
<tr>
<td>OTD$CREATE_DISABLE (Create Disable)</td>
<td></td>
</tr>
<tr>
<td>OTD$MUTEX (Mutex)</td>
<td></td>
</tr>
<tr>
<td>OTD$ALLOCATE (Allocate)</td>
<td></td>
</tr>
<tr>
<td>OTD$DEALLOCATE (Deallocate)</td>
<td></td>
</tr>
<tr>
<td>OTD$REMOVE (Remove)</td>
<td></td>
</tr>
<tr>
<td>OTD$DELETE (Delete)</td>
<td></td>
</tr>
<tr>
<td>OTD$SHUTDOWN (Shutdown)</td>
<td></td>
</tr>
</tbody>
</table>

The purpose of each of these fields is described below.

5.3.1.1  *otd$type Field (Static)*

This field contains a value indicating the type of object the OTD represents.

5.3.1.2  *otd$objhdr_listhead Field (Dynamic)*

This field contains the forward link to the first object header of this type, and the backward link to the last object header of this type. These links are used for consistency checking within the object architecture.

5.3.1.3  *otd$count Field (Dynamic)*

This field contains a count of the number of objects of the corresponding type that currently exist in the system. Modification of this field must be done using the RMALI instruction.
5.3.1.4 **otd$dispatcher_object_offset Field**

When nonzero, this object type supports the notion of wait. The value in this field is the offset of the dispatcher object (dispatcher objects are described in Chapter 4, The Kernel) from the start of the object body. Thus, when a wait operation is performed on the object, the executive adds the field contents to the address of the object header and issues a kernel wait operation on the dispatcher object.

The dispatcher objects that support waiting are:

- Events
- Semaphore
- Timer
- Thread
- Queue

These objects are created by providing the necessary structure in the object body and calling the kernel routine `k$initialize_xxxx`, where `xxxxx` is the type of dispatcher object to be initialized. Once initialized, these objects can then be manipulated by using other kernel routines. For more details on the kernel support provided, see Chapter 4, The Kernel.

Any object type which has a non-zero dispatcher field must allocate the object body from nonpaged pool. The wait routines which accept object types with non-zero dispatcher offsets call the kernel wait routines which operate at IPL 2 and cannot take page faults.

5.3.1.5 **otd$access_mask Field (Static)**

This field points to the supported access mask for use by the security routines in access validation. For more information, see the Chapter 10, Security and Privileges.

5.3.1.6 **otd$allocation_listhead_offset Field**

When nonzero, this object type may have objects allocated to it. For example, thread, process and job objects all support the notion of object allocation. Object allocation is described in Section 5.8.

5.3.1.7 **otd$create_disable Field (Dynamic)**

This flag may be set by the executive to prevent additional objects of this type from being created. It can be used to shut down the system in an orderly fashion.

Once set, this field may not be cleared. Therefore, access does not have to be interlocked. This flag is set when it contains a nonzero value.

5.3.1.8 **otd$mutex Field**

Provides synchronization for creation, deletion, and state changes among objects within a type.
5.3.1.9  otd$allocate Field

This field points to an allocate procedure. The allocate procedure is called in kernel mode at IPL 0 with the allocation mutex acquired. The address of the object body to be allocated and the allocation type are passed as input parameters.

The allocate procedure has the following type declaration:

```c
objc$object_allocate_procedure :
PROCEDURE (  
    IN object_body : POINTER anytype CONFORM;
    IN allocation_object_hdr : POINTER e$object_header;
);
```

The procedure `objc$null_allocate_procedure` is provided for use by object types which do not have an allocate procedure.

5.3.1.10  otd$deallocate Field

This field points to a deallocate procedure. The deallocate procedure is called in kernel mode at IPL 0 with the allocation mutex acquired. The address of the object body to be deallocated is passed as an input parameter.

The deallocate procedure has the following type declaration:

```c
objc$object_deallocate_procedure :
PROCEDURE (  
    IN object_body : POINTER anytype CONFORM;
    IN allocation_object_hdr : POINTER e$object_header;
);
```

The procedure `objc$null_deallocate_procedure` is provided for use by object types which do not have an deallocate procedure.

5.3.1.11  otd$remove Field

This field points to a remove procedure. The remove procedure is called when an object's object ID count field is decremented to 0. It is called in kernel mode, at IPL zero. It is passed two arguments, the address of the object body and the mode (user or kernel).

The remove procedure allows the object type-specific procedure to perform any necessary actions now that the ID of the object is removed. Once object ID count field is zero, it is impossible for a user to translate an object ID which points to this object.

The remove procedure has the following type declaration:

```c
objc$obj_remove_procedure :
PROCEDURE (  
    IN object_body : POINTER anytype CONFORM;
    IN access_mode : k$processor_mode;
);
```

The procedure `objc$null_remove_procedure` is provided for use by object types which do not have a remove procedure.
5.3.1.12 otd$delete Field

This field points to a delete procedure. The delete procedure is called when an object's pointer count field is decremented to zero. It is called in kernel mode at IPL zero. It is passed the address of the object body.

The delete procedure is responsible for manipulating object type-dependent data structures and deallocating the storage allocated for the object body extensions.

The delete procedure has the following type declaration:

```plaintext
obj$obj_delete_procedure :
PROCEDURE (   
    IN object_body : POINTER anytype CONFORM;   
); 
```

The procedure `obj$null_delete_procedure` is provided for use by object types which do not have a delete procedure.

5.3.1.13 otd$shutdown Field

This field points to a shutdown procedure. The shutdown procedure is called once when an object type is permanently removed from the system (presumably at system shutdown time). It is called by the executive in kernel mode at IPL 0 with ASTs enabled. The purpose of this routine is to provide a way to perform object type-specific shutdown operations. Among other things, this provides the opportunity to dump any statistics that may have been gathered relating to the object type.

The parameters passed to this routine include the address of the OTD, and the address of a callback routine that can be used to output information to a shutdown log file. This routine must be provided even if it simply returns without performing any actions.

The shutdown procedure has the following type declaration:

```plaintext
obj$obj_shutdown_procedure :
PROCEDURE (   
    IN otd_hdr : POINTER e$object_header;   
    IN log_procedure : e$log_procedure;   
); 
```

The procedure `obj$null_shutdown_procedure` is provided for use by object types which do not have a shutdown procedure. procedure.

5.4 Object Header

The object header is a fixed-format data structure that contains object type-independent data. This header is used by the executive without necessarily knowing the type of object it is accessing.

```plaintext
e$object_header : RECORD
    objhdr$pointer_count : integer;
    objhdr$object_id_count : integer;
    objhdr$type : e$object_type;
    objhdr$otd : POINTER e$object_header;
    objhdr$link : e$linkedList;
    objhdr$container : POINTER e$object_header;
    objhdr$level : e$level;
    objhdr$index : e$index;
    objhdr$seq_number_low : e$seq_number;
    objhdr$seq_number_high : e$seq_number;
```

Object Architecture 5-15
Digital Equipment Corporation - Confidential and Proprietary
Restricted Distribution

$objhdr$dispatcher_object : POINTER anytype CONFORM; !# POINTER k$dispatcher_obj
$objhdr$name : POINTER $object_name_block;
$objhdr$owner : $identifier;
$objhdr$accl : POINTER e$access_control_list;
$objhdr$allocation_block : POINTER e$object_allocation_block;
$objhdr$access_mode : k$processor_mode;
$objhdr$transfer_flag : bit; ! 0 - transfer not allowed
                     ! 1 - transfer allowed
$objhdr$reference_inhibit_flag : bit; ! 0 - reference ids allowed
                                ! 1 - reference ids not allowed
$objhdr$temporary_inbit_flag : bit; ! 0 - permanent
                              ! 1 - temporary
$objhdr$temporary_operation : bit; ! 0 - make temporary
                                ! 1 - mark temporary

END RECORD;

5.4.1 Object Header Data Structure

Each of the fields in the object header are defined to be either static or dynamic. If a field is defined as static, its value is established during object creation, and remains constant for the life of the object. Static fields may be accessed any time the pointer count field has a nonzero value (see the description of the objhdr$pointer_count field in Section 5.4.1.1). However, if a field is defined to be dynamic, it is subject to modification. Access to dynamic fields is controlled by a protocol that is specific to each field. The specific protocol indicating when each dynamic field may be accessed is defined with the description of the field.

5.4.1.1 objhdr$pointer_count Field (Dynamic)

Whenever an access to an object will span a period of time that can not be protected by use of mutex locks on appropriate data structures, the pointer count field must be incremented. In particular, an address of an object may not be used to regain access to the object unless the pointer count has been previously incremented.

The pointer count represents the number of pointers that have been taken out on the object, plus one for a nonzero object ID count. When an object ID is translated by an object service routine, the pointer count is incremented by one. The pointer count for an object signifies the number of reasons the storage for an object should not be deallocated.

To increment this field, the directory-level mutex must first be held (see Section 5.5.3.4, below). This is necessary to avoid race conditions with the object ID being deleted.

When an object ID, object container pointer, or reference object ID is deleted, the object ID count of the principal object is decremented. If the resultant object ID count is zero, the object type-specific remove routine is called to initiate any object specific removal action such as canceling I/O. A zero object ID count also causes the pointer count to be decremented, as does the dereferencing of a pointer. When the pointer count of an object reaches zero, the object type-specific delete routine is called, and the object header storage is deallocated.

Note, if the object ID count is zero then no object container has a pointer to that object header.

One of the major goals of the lock strategy for objects is to allow the pointer count to be decremented (using the RMALI instruction) without having to hold a lock to do so. This allows low overhead dereferencing of objects on such operations as I/O, wait, and the deleting of a reference object.

If the pointer count is zero and the object ID count is nonzero, a bug check is issued.
Figure 5-4: Object Header Format

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>OBJHDR$POINTER_COUNT</td>
<td>Pointer Count</td>
</tr>
<tr>
<td>OBJHDR$OBJECT_ID_COUNT</td>
<td>Object ID Count</td>
</tr>
<tr>
<td>OBJHDR$TYPE</td>
<td>Object Type</td>
</tr>
<tr>
<td>OBJHDR$OTD</td>
<td>Object Type Descriptor</td>
</tr>
<tr>
<td>OBJHDR$LINK</td>
<td>Linked List</td>
</tr>
<tr>
<td>OBJHDR$CONTAINER</td>
<td>Container</td>
</tr>
<tr>
<td>OBJHDR$LEVEL</td>
<td>Level</td>
</tr>
<tr>
<td>OBJHDR$INDEX</td>
<td>Index</td>
</tr>
<tr>
<td>OBJHDR$SEQ_NUMBER_LOW</td>
<td>Sequence Number Low</td>
</tr>
<tr>
<td>OBJHDR$SEQ_NUMBER_HIGH</td>
<td>Sequence Number High</td>
</tr>
<tr>
<td>OBJHDR$DISPATCHER_OBJECT</td>
<td>Dispatcher Object</td>
</tr>
<tr>
<td>OBJHDR$NAME</td>
<td>Name</td>
</tr>
<tr>
<td>OBJHDR$OWNER</td>
<td>Owner</td>
</tr>
<tr>
<td>OBJHDR$ACL</td>
<td>ACL</td>
</tr>
<tr>
<td>OBJHDR$ALLOCATION_BLOCK</td>
<td>Allocation Block</td>
</tr>
<tr>
<td>OBJHDR$TEMPORARY_OPERATION</td>
<td></td>
</tr>
<tr>
<td>OBJHDR$ACCESS_MODE</td>
<td>Access Mode</td>
</tr>
<tr>
<td>OBJHDR$TRANSFER_FLAG</td>
<td>Transfer Flag</td>
</tr>
<tr>
<td>OBJHDR$REFERENCE_INHIBIT_FLAG</td>
<td></td>
</tr>
<tr>
<td>OBJHDR$TEMPORARY_FLAG</td>
<td>Temporary Flag</td>
</tr>
</tbody>
</table>

5.4.1.2 objhdr$object_id_count Field (Dynamic)

The object ID count represents the number of object IDs or container directory index table pointers that can be used to refer to the object. For all objects except container objects, this is the number of object IDs that refer to the object. For container objects, this is the number of container directories that can refer to this object container (note that the object ID itself is not counted). Hence, for object containers, the object ID count is one unless the container is a display container.
The object ID count signifies the number of reasons that the remove action for an object should not be invoked.

If an object is marked as temporary (as discussed in Section 5.4.1.18, below), then it must have at least one reference to it.

To increment this field, the mutex field of the object container in which the object resides and the directory-level mutex must first be acquired (see Section 5.5.3.4, below). This is necessary to ensure that the object cannot be deleted while another reference is being established.

Note that this field is never directly incremented or decremented; the executive routines for object support operate upon this field.

5.4.1.3 objhdr$type Field (Static)

This field contains the integer value of the object type. When an object ID is translated to an object header, the type field is compared to the desired object type.

5.4.1.4 objhdr$otd Field (Static)

The object type descriptor field contains a pointer to the object type descriptor (OTD) for this particular object type. The executive and the object type's service routines use this field to use information or routines associated with the OTD. Note that there is exactly one OTD per object type defined in the system.

The OTD is described in detail in Section 5.3.1.

5.4.1.5 objhdr$link Field (Dynamic)

This field contains the forward link to the next object header of this type, as well as the backward link to the previous object header of this type. It is used for debugging purposes.

5.4.1.6 objhdr/container Field (Dynamic)

The container field is a pointer to the object container in which the object resides.

This field is dynamic only because the object may be transferred to a new object container. Such a transfer would result in the assignment of a new object ID. This field is zeroed when the principal object ID is removed from its object container.

This field may only be accessed by a thread holding both the directory level mutex and the mutex within the object container.

5.4.1.7 objhdr$level Field (Dynamic)

The level (system, job, process) at which the object's principal ID exists. This field is unaffected if the principal ID is deleted.

5.4.1.8 objhdr$index Field (Dynamic)

This field is the index into the object array of the container object specified in the container field for this object.

This field is dynamic only because the object may be transferred to a new object container. Such a transfer would result in a new object ID being assigned. This field may only be accessed by a thread holding both the directory-level mutex and the mutex within the object container.
5.4.1.9 objhdr$seq_number_low and objhdr$seq_number_high Fields (Dynamic)

When an object ID is translated into the address of an object, the value in the object ID's sequence field is compared to the value in the object header's sequence field. If they are not identical, then the object ID has been reallocated, and the one specified for translation is no longer valid.

This field is dynamic only because the object may be transferred to a new object container. Such a transfer would result in the assignment of a new object ID. This field may only be accessed by a thread holding both the directory-level mutex, and the mutex within the object container.

Note that it is possible to construct the object ID given an object header. Two parts of the object ID are in the object header, the sequence number and the object index. The container field points to the object header of a container. The index field in that container is the directory index. The level value in the container directory body is the level portion of the ID.

5.4.1.10 objhdr$dispatcher_object Field (Static)

If the object type allows wait operations this field points to the dispatcher object to be used in the kernel wait operation. If the object type does not support the notion of wait, this field has the value nil.

5.4.1.11 objhdr$name Field (Dynamic)

This field is used to associate a name string with an object. If the object has an associated name, this field contains a pointer to an name definition block which contains the name. Otherwise, this field has a value of zero.

When the object ID is deleted, the associated name, if any, must also be deleted.

The level directory, the object container, and the type-specific mutex must all be acquired in order to modify this field.

See Section 5.5.5 for a complete description of how object names are managed.

5.4.1.12 objhdr$owner Field (Static)

The identifier of the owner of the object. This field is used by the security access validation routines. For more information see Chapter 10, Security and Privileges.

5.4.1.13 objhdr$aci Field (Dynamic)

The format and use of this field is defined in Chapter 10, Security and Privileges.

This field is a pointer to the ACL for the object. A value of nil indicates that no ACL exists for the object.

5.4.1.14 objhdr$allocation_block (Dynamic)

The field is used by the object translation routines to ensure that the thread has access to the object. In order to translate an object ID to an object header, the requesting thread must have access to the object as determined by the ACL, and must be in the allocation class of the object. Object allocation and allocation classes are discussed in Section 5.8.

The allocation mutex and the type mutex must be acquired in order to modify this field.
5.4.1.15 objhdr$access_mode Field (Static)
This field contains the owner mode (kernel or user) of the object. Kernel-mode objects may not be created in user owned containers. Mode is also used to verify access to the object by the access validation procedures as described in Chapter 10, Security and Privileges.

5.4.1.16 objhdr$transfer_action Field (Static)
When clear (false) this flag indicates that the object cannot be transferred to another container. The transfer action field state is determined at object creation from the transfer state flag in the OTD. This flag is examined when an attempt is made to transfer an object using the exec$make_temporary service or job/process creation services.
Transfer action is discussed in Section 5.7.8.

5.4.1.17 objhdr$reference_inhibit Field (Dynamic)
When set (true) an attempt to create a reference ID to this object results in a failure.

5.4.1.18 objhdr$temporary_flag Field (Dynamic)
When set, this flag indicates the object is a temporary object. A temporary object has the property that when all reference objects to the object are deleted, then the object itself is also deleted. Therefore, when the object ID count field is decremented to one, this field is checked. If it is set, then object deletion may be invoked. See Section 5.7.7 for more information.
The level directory, the object container, and the type-specific mutex must all be acquired in order to modify this field.

5.4.1.19 objhdr$temporary_operation Field (Dynamic)
This flag describes the type of operation that was used (make temporary or mark temporary) to make the object temporary. This flag is only meaningful when the objhdr$temporary_flag is set.

5.4.2 object_body Record
The purpose and use of this record is left to the designer of a particular object type.
The object_body record is used to store object type information. The format and meaning of fields within this record are entirely the responsibility of the object's designer. It is also up to the designer of the object type to define the protocols for accessing various fields within this record.
In many cases, the object_body record can be allocated in a single block (with the object header) and is sufficient to hold all the object type-specific information associated with an object. Certainly this is true for simple objects like events. For more complex objects, like object containers, it may be necessary to have other data structures associated with the object.
The content and management of these additional data structures is entirely the responsibility of the object's designer. They must be connected in some fashion to the object_body record so that they are deleted when the object is deleted.
Object bodies are allocated from paged or nonpaged pool. If the object body contains any kernel dispatcher objects or kernel control objects it must be allocated from nonpaged pool.
Object bodies should not contain the object IDs of any objects. If a need exists to refer to another object a pointer to that object's body should be used. This prevents dependencies on other object IDs being valid.
5.5 Object Container Data Structures

Object containers are objects. Therefore, their primary data structures are:

- Object container object type descriptor (OTD)
- Object container object header
- Object container body

To track objects and their associated names, object container bodies have two primary associated data structures. These data structures are:

- Object array
- Name table

Figure 5–5 illustrates the relationships between these object container data structures.

Figure 5–5: Object Container Data Structure Relationships

![Diagram of object container data structure relationships]

In addition to these data structures, object names are stored in name definition blocks.

The exact data structures used to support object names, and is subject to change following performance analysis yet to be performed. The algorithms for manipulating the ultimate data structures will have to be implemented to match the data structures, and so, are also subject to change.

The aspects of the current design that are not expected to change are:

- The name field of the object header points to a data structure containing the case-sensitive name of the object. This pointer provides the link necessary to delete the object name when the object is deleted.

- A field in the object container body (currently called the name table field) is used to locate the name management data structures for name assignment and translation purposes.
5.5.1 Container Directories as Compared with Object Containers

Container directories are similar in structure to object containers. The object header is the same, except for type, and the object body is nearly identical. The only differences are:

- Container directories do not have a mutex in their object bodies.
- Only container directories have a display index field.

There are some restrictions placed on the container directory that are enforced by the executive object manipulation routines. They are:

- The only objects that can be pointed to by (contained within) a container directory object are object containers.
- The only names that can be stored in a container directory are the names of the object containers that are pointed to by the container directory and logical names.

5.5.1.1 Object Container OTD Remove Procedure

When an object container is deleted, the OTD remove procedure removes the ID of each object within that container by calling the routine `obj$remove_obj_from_container`.

5.5.2 Object Container Object Header

The fields of the object container's object header are initialized and used as defined in Section 5.4. Note that the object container is an object, and is pointed to by the first pointer in the object index table.

Also, the object index field of the object header has a value of zero for a container directory. Object containers contain the value corresponding to the object index array element in the container directory, which points to this object container.

The following restrictions exist for object containers:

- An object container (or container directory) cannot be allocated.
- An object container (or container directory) cannot be temporary.

5.5.3 Object Container Body

The format of the object container's body is shown in Figure 5–6.

**Figure 5–6: Object Container Body Data Structure**

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CON$OBJTBL</td>
<td>(Pointer to Object Array)</td>
</tr>
<tr>
<td>CON$OBJNAMTBL</td>
<td>(Pointer to Name Table)</td>
</tr>
<tr>
<td>CONDIR$DISPLAY_INDEX</td>
<td>(Display Index – found only in container directories)</td>
</tr>
<tr>
<td>OBJCON$MUTEX</td>
<td>(Mutex – Found only in object containers)</td>
</tr>
</tbody>
</table>

The purpose of each of these fields is described below.

The object array field is a pointer to the object container's object array. The object array is described in Section 5.5.4, below.
Note that the first object pointer of this table points to the object header for this container.

5.5.3.1 con$objtbl Field
This field contains a pointer to the object array.

5.5.3.2 con$objnamtbl Field (Static or Dynamic, Depending on Implementation)
The name table field is a pointer to the table used to translate names for objects and logical names in this object container. The name table is described in Section 5.5.5, below.

5.5.3.3 condir$display_index Field (Static)
This field is only in the container directory.

The purpose of the display index field supports the capability of a process to access objects in its ancestors' display object containers (as discussed in Section 5.2.6.5).

The first element in the object array (element 0) always points to its own object header.

In all cases except process level, the second element (element 1) always points to the first object container for that level. In these cases, the display index field contains the value one.

However, for process level container directories, the process's ancestors' display containers are pointed to by the second through nth elements (elements 1, 2, 3, etc.). In this case, the display index field contains the element number containing the current process's display object container. This element is then followed immediately by private process object containers. Figure 5–7 illustrates the use of the display index field in a process level container directory.
5.5.3.4 objcon$mutex Field (Static)

The mutex field is a kernel mutex which is operated on by issuing calls to procedures within the kernel. These procedures are described in Chapter 4, The Kernel.

This mutex is used to synchronize access to certain fields in the object header.

5.5.4 Object Array Data Structure

The primary function of the object array is to hold a pointer to each created object in the object container. The format of the object array is shown in Figure 5–8.

The purpose of each of these fields is described below.

5.5.4.1 objtbl$max_index Field

This field indicates the current length of the object array by providing a count of the number of elements in it minus one. The value in this field is the maximum valid array index value.

This field value has an architectural limit of 1,048,575 (20 bits).
Figure 5–8: Container Object Array Format

| OBJTBL$MAX_INDEX (Object Array Length) |
| OBJTBL$COUNT (Count) |
| OBJTBL$NEXT_FREE (Next Free) |
| OBJTBL$TABLE (Object Array) |

Figure 5–9: Address of Object Header

<table>
<thead>
<tr>
<th>31</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADDRESS OF OBJECT HEADER</td>
<td></td>
</tr>
</tbody>
</table>

5.5.4.2 objtbl$count Field

This field is initialized to 1 when the object container is created, to represent the fact that this object container ID is in the array. It is then incremented each time an object ID within this object container is created. Similarly, it is decremented when an object ID in this object container is removed from the container as part of deletion.

5.5.4.3 objtbl$next_free Field

When created, an object container has a pointer to itself as the first element of the object array, and the remaining elements are free (unused). To facilitate quick allocation of free object array elements, they are organized into a linked "free list".

The next free field is the listhead of the free list, and contains the index of the next object array element to be allocated. The free list link field of the first element contains the index of the second, which contains the index of the third, and so on. The last element has a free list link value of zero to indicate the end of the list has been reached.

5.5.4.4 objtbl$table Field (Object Array)

Object array elements are used for two purposes. First, each object in the object container has an element in the object array that is associated with it. Such an element contains a pointer to the header of the object. In this case, the object element is referred to as "in use." The format of an object array element when it is in use is discussed below.

As discussed in Section 5.2.4, each time an object ID is allocated, it receives a new sequence number. This sequence number high field is associated with an object array element, and is stored within the element when the element is not in use. When the element is in use, the sequence number is moved to the sequence high field of the associated object's header. When the object is removed from the object container, the sequence number is incremented, reset to zero on overflow, and moved back into the object array element.
The format of a free object array element is:

The free list link field is used to link all unused object array elements together on a single list for quick allocation (as discussed in the next section).

Notice that the most significant bit (MSB) of object array elements may be used to determine whether the element is currently in use or free. If the MSB is 1, then the element contains an address in system address space and the element is currently in use. If the MSB is 0, then the element is currently free and contains the next sequence number to use and a link to the next element on the free list.

Object array element index values begin at 0 and extend to the value in the $objtbl$max_index field.

5.5.5 Name Table Data Structure

The purpose of the name table is to track both logical and object names within an object container.

When an object container is created, a name table must also be allocated, and the address of the name table must be stored in the name table field of the object container's body. The name table contains both object names which translate to an object ID and logical names which translate to an equivalence string.

The name table provides for a mechanism to translate names to name definition blocks. The actual implementation details of the name table need not be defined, only the functions provided by the name table are necessary. For example, the name table could be implemented as a hash table or a binary tree.

5.5.5.1 Name Definition Block

The name definition block contains information describing the name associated with a particular object or logical name. This information includes the type, mode, object ID, and other information about the name.

5.5.6 Object Naming Principles

Names are qualified by object type and mode. The combination of object type, mode, and name string is unique within a single object container.

For example, you can have two widget objects named SYSSZEBRA in a single object container, but only if one is a user-mode object, and the other is a kernel-mode object. Likewise, you can have two user-mode objects named DEVICE_CONTROL in the same object container, but only if they have different object types.

Object names are limited to 255 characters in length. All object names are stored exactly as received (that is, case-sensitive) and may be looked up (translated) either case-sensitive or case-blind.

When a case-sensitive name translation is requested, either one or zero object IDs is returned. However, when a case-blind name translation is requested, many object IDs may match the name. To illustrate this, consider two objects, one named "pQr" and the other named "pQR". If a case-sensitive translation of "PQR" is requested, no object IDs is returned. If a case-blind translation of the same string, "PQR", is requested, then only the first name found which matches is returned.
Software which intends to use the case-blind feature of name translation is responsible for ensuring consistency. For example, all names strings could be converted to upper case before calling the object service routines. This prevents multiple names from matching in a case-blind search.

5.5.7 Naming

Names are used in the object architecture to label objects and equivalence strings. The same naming mechanism is used when a name is applied to either an object or an equivalence string. The only difference is the translation of an object name yields an object ID, while the translation of a logical name yields an equivalence string.

A name, like an object, exists within an object container. It is further qualified by the type of the object it names (logical names have the special object type of 0), and the mode in which it was created (user or kernel).

Through use of object name translation services a name can be searched for in any object container to which the user has access. The object name translation services provide recursion. See the description of exec$translate_object_name for more details.

The logical name translation services do not automatically provide recursion, rather it is the responsibility of the facility using the logical name translation services to provide for recursion, search lists, and parsing. This allows for a facility such as RMS to have different logical name evaluation rules.

5.5.7.1 Logical Names

Logical names are names that translate to equivalence strings. Each logical name can have up to 127 equivalence strings. The term "multivalued logical name" means a logical name with more than one equivalence string. Each equivalence string can be up to 255 bytes in length, and each equivalence string can have any, all, or none of the following attributes:

- **Terminal**—The equivalence string is not to be subjected to further logical name translations.
- **Concealed**—The application should not return the equivalence string to the user.

Logical names can have the following attributes:

- **No show**—The logical name should not be displayed in general show logical name services.
- **No alias**—The logical name cannot be duplicated in this table at an outer access mode. If another logical name with the same name already exists in the table at an outer access mode when a new logical name with no alias is created, it is deleted.
- **Confine**—The logical name is not copied from the process to its spawned subprocesses. This applies only to logical names at the process level in private object containers.

5.5.7.2 Object Services Providing Name Support

The following object services provide generalized name support.

- exec$set_object_name
- exec$translate_object_name
- exec$clear_object_name
- exec$create_logical_name
- exec$translate_logical_name
- exec$delete_logical_name

These routines are described in the Internal System Services Manual.
5.5.7.3 Container Directory Names

The three container directories to which the process control block points are named. The process container directory is named exec$process_container_directory, the job container directory is named exec$job_container_directory, and the system container directory is named exec$system_container_directory.

Each of the container directories has a name table (see Figure 5–2). This name table contains object container names which translate to the ID of an object container, and logical names, which translate to equivalence strings. When an object container is created, the name, if specified, is stored in the container directory's name table.

By creating "multivalued" logical names in the container directory name table, a search list for object containers can be created.

5.5.8 Object Name Translation

Object name translation involves taking a specified name and object type, and returning the related object ID (if any). The name table performs this function such that given a name, all name definition blocks containing that name are searched for the specified type.

The executive object support routine obj$translate_object_name provides the mechanism for object service routines to translate an object name to an ID.

PROCEDURE obj$translate_object_name (  
    IN container_id : e$object_id;  
    IN name : string (*);  
    IN object_type : e$object_type;  
    IN access_mode : k$processor_mode;  
    IN case_blind : boolean;  
    OUT object_id : e$object_id; ) 
    RETURNS STATUS;

By specifying an OBJECT_TYPE of zero, the first object found which matches the name is returned. obj$translate_object_name performs the following:

1. Acquires the directory-level mutex.
2. Translates the ID of the object container to a pointer to an object header and check access for read.
3. Ensures that the object header is a container or a directory.
4. Acquires the container mutex and releases the directory mutex if the object type is container.
5. Looks up the specified name in the container name table that matches the specified object type and mode.
6. Raises an error condition if the specified name is not found.
7. Returns the object ID which corresponds to the name if the name is found.
8. Releases the mutex.
5.5.9 Initializing a Process Level Container Directory

Container directories at system and job levels are always initialized to be empty (that is, they initially do not point to any object containers other than themselves). Container directories at the process level, however, must be initialized so that all of the process's ancestors' display containers are accessible. Also, in order to preserve the object ID of objects in its ancestors' display containers, it is necessary that they be pointed to by the same object array element in each process's container directory. To achieve this, process-display object containers are always the second through nth entries of a process level container directory.

A new process level container directory is initialized by copying elements starting at element one, and continuing through the value specified in the display index from the parent process's container directory. These entries are followed immediately by the new process's display and then private object containers. Figure 5–11 illustrates the initialization of a new (child) process's container directory.

![Diagram](image)

**Figure 5–11: Process Level Container Directory Initialization**

In this example, the condir$display_index field of the parent's container directory indicates the last object container which must be copied to the new process's container directory. The new process's condir$display_index field is then updated to point to its own display container.

If the user deletes a display container, the table element within the container directory that pointed to the display container cannot be reused. If this were allowed, a private container could be copied as a display container.

5.5.10 System Global Variables and Data Structures

This section identifies the system global variables and data structures required to support the object architecture. The data structures that are accessible via known system locations include:

- System level container directory (see Section 5.2.8)
- System level directory mutex (see Section 5.2.6.2)
- Allocation mutex
5.5.10.1 OTD Objects

Each object of type OTD is created in a reserved system level object container. When a set of object services for an object type is loaded, the object service image is called at its initialization entry point. During the initialization operations, the corresponding OTD object must be created by calling the obj$create_odb service.

The obj$create_odb procedure has the following declaration:

```plaintext
PROCEDURE obj$create_odb(
    OUT otd_object_id : exec$object_id;
    IN object_item_list : ITEM LIST CONFORM OPTIONAL;
    IN transfer_state : boolean;
    IN dispatcher_offset : integer;
    IN shutdown_procedure : PROCEDURE e$shutdown_procedure;
    IN remove_procedure : PROCEDURE obj$remove_procedure;
    IN delete_procedure : PROCEDURE obj$delete_procedure;
    IN allocate_procedure : PROCEDURE obj$allocate_procedure;
    IN deallocate_procedure : PROCEDURE obj$deallocate_procedure;
    OUT otd_value : integer;
);
```

At system shutdown, the operating system searches the OTD object container to locate each object type:

Each object type is unloaded by:

1. Setting the OTD’s create disable flag (to prevent additional objects of that type from being created).
2. Calling the shutdown routine specified in that object type’s OTD when all objects of a particular type have been deleted.

5.5.10.2 Allocation Mutex

The allocation mutex is a system-wide mutex which is acquired when an object’s allocation is checked, modified, or deleted.

5.6 Relative Ordering of Object Architecture Mutexes

The following lists illustrates the relative ordering of object architecture mutexes. The mutexes are listed in order of lowest value to highest value.

- Process level container directory mutex
- Process level object container mutex
- Job level container directory mutex
- Job level object container mutex
- System level container directory mutex
- System level object container mutex
- OTD type mutex
- Allocation mutex
5.7 Object Creation

5.7.1 Creating an Object

Each type of object is supplied with one or more service routines which create the object type. For example, a FRAMITZ object might be created with a create_framitz object service.

Object creation routines should have the following procedure declaration:

    PROCEDURE exec$create_xxxxx ( !where xxxxxx is the object name
        OUT xxxxxx_id : exec$object_id;
        IN object_independent_items : exec$object_parameters CONFORM OPTIONAL;
        IN object_dependent_items : exec$item_list CONFORM OPTIONAL;
    ) RETURNS STATUS;

The object independent items consist of the following:

- Object ID of object container in which the new object is to be created
- Name to associate with the object
- Access control information

The format and processing of object ownership and access control information is discussed in Chapter 10, Security and Privileges.

In addition to these standard parameters, object creation services may accept type-specific parameters. For example, an event creation service may accept an initial state of the event. The name of an object creation routine and the parameters allowed by that routine are specified by the designer of the object type.

5.7.2 Object Names

When creating an object, it is possible to assign a character string name to the object. This is particularly useful for objects that are to be shared. It allows one process to create an object (assigning it a name) and then, sometime later, another process can locate and access the object via its name.

Also, object names provide "create if" semantics. An attempt to create an object of same type, mode and name in the same container will fail, but the ID of the object which had the same name, type and mode is returned as is a warning status code.

5.7.3 Create Object Algorithm

The user calls an exec$create_object service specifying the container ID, ACL, returned object ID, and optionally a name. Other object type-specific arguments may be required.

Every create object service performs the following steps:

1. Probes the user's arguments.
2. Calls obj$create_initilize_object passing in the following:
   - Object ID of OTD object
   - Pool type (paged or nonpaged)
   - The number of bytes of storage to allocate with the object header for the object body
   - The object name, if any
   - Ownership mode (kernel or user)
   - ACL
   - Transfer flag state
The `obj$create_initialize_object` routine locates the specified object type and allocates the storage.

At this point the object header has been created and initialized. The pointer count field is one, the object ID count and the container fields are zero.

3. Initializes the object body. This action is object type-dependent, for example, if the object supports waiting, a call to `k$initialize xxxx`, where `xxxx` is a dispatcher initialization routine, is issued.

At this point the object has been created but has no ID.

4. Attempts to add a pointer to the object header in the specified object container by calling `obj$insert_object_into_container` with the address of the object header, the container ID, the mode, and the object ID to be returned.

`obj$insert_object_into_container` does the following:
- Acquires the specified directory-level mutex.
- Translates the ID of the object container to a pointer to an object header, and checks for write access. Ensures that the object header is a container.
- Checks the transfer state of the container. If the container is transferable, and the object is not transferable, `obj$insert_object_into_container` returns an error status to the caller.
- Acquires the container mutex.
- Releases the directory mutex.
- If an object name was specified, `obj$insert_object_into_container` checks to see if the same name already exists within this container with the same type and mode. If a name conflict occurs, `obj$insert_object_into_container` calls the OTD delete routine, releases the container mutex, and returns the ID that corresponds to the collided name.
- The next free field in the object container's body is examined.
  - If the next free field is nonzero `obj$insert_object_into_container` does the following:
    * Removes the element specified in the next free field from the free list, and creates a new next free element.
    * Constructs the sequence number high field by using the value found in the object array, stored in the object header, and the address of the object header is stored in the element.
    * Constructs the object ID from the element number, the sequence number high, a random 10-bit number for the sequence number low, and the object index field of the object container.
      The object header is completed.
    * Sets the object ID count to 1, the container field points to the header of the object container, and so on.
    * Increments the create count field in the OTD for this object type using the RMALI instruction.
    * Increments the total ID field in the object container.
    * Releases the object container mutex, and returns the object ID to the user.
  
  - If the next free field is zero, `obj$insert_object_into_container` does the following:
    * Checks some resource quota to see if the object container can be expanded. If the object container cannot be expanded, `obj$insert_object_into_container` calls the OTD delete procedure. The container mutex is released, and a container full error is returned to the user.
Allocates a new (larger) object array data structure.
* Copies the contents of the old object array to the new object array.
* Links all new object array entries beyond the length of the old object array into the free list.
* Sets the new object array's length.
* Changes the object array field in the object container to point to the new object array.
* Deallocates the old object array data structure.
* Creates the object ID as in the above case with a nonzero objtbl$next_free field.

At this point, if the object header is successfully added to the specified object container, both object ID count and pointer count fields contain the value one.

5. Adds the name, if any, to the object container name table. The container, the sequence number and object index fields are initialized.

6. Returns the object ID and status to the user.

If an ID cannot be created, the object type-specific delete procedure found in the OTD is invoked. Note that the container field and object ID count in the object's header were zero prior to the object's deletion. An error status is returned.

Once the object has been inserted in the specified container, the object creation routine may not reference the object body without calling the obj$reference_object_by_id routine.

The following status values could be returned from obj$insert_object_into_container:

- Invalid object container—Returned object ID is set to zero (error).
- Container full—Returned object ID is set to zero (error).
- Resource limit exceeded—Returned object ID is set to zero (error).
- Name collision—Returned object ID is set to the ID of the object with the same name and type (warning).
- Success—Returned object ID points to newly created object.

These rules cause all objects to be created in a uniform way. If two users attempt to create an object in the same container with the same name, type, and mode, only one is created, yet both users get returned a valid ID. One user gets a success code indicating the object was created, the other gets a warning code indicating the object already exists.

5.7.4 Object Modes

When creating an object, its mode (user or kernel) is established. Any object creation routine which is called from user mode creates a user-mode object.

Object creation routines which are called from kernel mode allow the mode to be specified as an argument. Object service routines that are called from user mode have the argument supplied by the jacket routine, which calls the corresponding entry point of the routine with the mode argument supplied.

This capability allows executive software to create objects that are either inaccessible to user mode or cannot be deleted from user mode.

Note that it is not possible to create an object owned by kernel mode in an object container owned by user mode. If this were allowed, the user would be able to delete the kernel mode object ID by deleting the container.
5.7.5 Object Access Protection

Object access protection is provided by a combination of user identifiers and access control lists (ACLs), and mode (user or kernel). This access protection information is stored for each object in its object header, and is processed by system-wide access validation routines without taking the object type into consideration.

Each object service routine is required to request access validation each time an object is accessed.

The exact format of the access protection information and procedures for performing access validation are provided in Chapter 10, Security and Privileges.

5.7.6 Object ID Translation

The executive procedure obj$reference_object_by_id provides translations from an object ID to a pointer to an object body. The procedure has the following declaration:

```
PROCEDURE obj$reference_object_by_id (  
   IN object_id : e$object_id;  
   IN object_type : e$object_type;  
   IN access_mode : k$processor_mode;  
   IN desired_access : e$access_type;  
   OUT object_body : POINTER anytype CONFORM;  
) RETURNS STATUS;
```

In order to translate an object ID to the address of the associated object, obj$reference_object_by_id performs the following steps:

1. Uses the level number as an index into the process control block to locate the corresponding level directory mutex.
2. Acquires the directory mutex.
3. Uses the level number as an index into the process control block to locate the corresponding container directory.
4. Compares the container directory index field to the size of the table to ensure that index is within the object array.
5. Uses the container directory index field to index into the object array.
6. Checks the resulting value to ensure that it is the address.
7. Uses the resulting system address to locate the object container header for the object container.
8. Compares the object index field to the size of the table to ensure that index is within the object array.
9. Uses the object index field to index into the object array.
10. Checks the resulting value to ensure that it is the address of an object.

At this point, the address found in the object array is the address of the object header.
11. Compares both sequence number fields in the object header to the sequence number fields in the object ID. If they are not identical, that object ID is not valid.
12. Checks the object type to ensure this object is of the desired type.
13. Validates the intended access to the object by calling the security procedure e$validate_access.
14. Releases the directory mutex and returns the error status indicated by e$validate_access to the user if access is denied.
15. Does not validate allocation access if the intended access is show.
16. Checks to see if the object is allocated by examining the allocation information field in the object header.

17. If the allocation information field is not nil, then \texttt{obj$reference_object\_by\_id} performs the following steps:
   
   \begin{itemize}
   
   \item Acquires the allocation mutex.
   
   \item Checks the allocation information field in the object header again. If the field is nil, \texttt{obj$reference_object\_by\_id} releases the allocation mutex and continue as if the field was always nil.
   
   \item Checks to ensure that the current thread is within the allocation class by calling the procedure \texttt{e$validate\_allocation}.
   
   \item Releases the allocation mutex.
   
   \item Releases the directory mutex, and \texttt{obj$reference_object\_by\_id} returns the error indicated by \texttt{e$validate\_allocation} to the user if allocation does not match.
   
   \end{itemize}

18. Increments the pointer count field in the object header so the object cannot be deleted while a pointer to the object is held.

19. Releases the directory mutex, stores the pointer to the object body, and returns to the caller.

5.7.7 Object Deletion

The user calls the \texttt{exec$delete\_object\_id} service specifying the object ID of the object to delete.

The \texttt{exec$delete\_object\_id} routine performs the following steps:

1. Probes the user's argument.

2. Calls \texttt{obj$remove\_obj\_from\_container} with the source ID and the mode of access.

The \texttt{obj$remove\_obj\_from\_container} routine performs the following steps:

1. Acquires the source container directory-level mutex.

2. Acquires the source container mutex.

3. Translates the source object ID to a pointer.

4. Acquires the type-specific mutex.

5. If address of the source container matches the container field in the source object header, then \texttt{obj$remove\_obj\_from\_container} zeros the container field and removes the object's name (if any) from the name table (in this case, this is the principal ID).

6. Decrements the object ID count of the source object.

7. Decrements the total ID field of the object container.

8. The \texttt{obj$remove\_obj\_from\_container} routine performs the following steps if the resultant object ID count is zero:
   
   \begin{itemize}
   
   \item Releases the type-specific mutex.
   
   \item Checks the allocation block field in the object header. If the field is not nil \texttt{obj$remove\_obj\_from\_container} does the following:
   
   \begin{itemize}
   
   \item Acquires the allocation mutex.
   
   \item Ensures the allocation block field is not nil.
   
   \item Calls the type-specific deallocate routine, passing in the address of the object body and the allocation type.
   
   \end{itemize}
   
   \end{itemize}

\textit{Object Architecture 5–35}
Cleans the allocation pointer in the object's header.

- Unlinks the object allocation block and calls `obj$dereference_object`.
- Releases the allocation mutex.
- Returns the storage occupied by the object allocation block to the system.

- Removes the object ID and name from the source container. This frees the slot, updates the sequence number, and links it into the free list.
- Releases the source container directory level and the source container mutexes.
- Calls the type-specific remove object ID routine.
- Calls `obj$dereference_object` to decrement the pointer count.
- Returns to the caller.

9. If the resultant object ID count is one, the source object is temporary, and the container field in the source object's header is not zero, then `obj$remove_obj_from_container` performs the following steps:
   - Constructs the object ID of the principal object.
   - Deletes object ID and name from source container.
   - Releases the source container directory level, the source container, and the type-specific mutexes.
   - Calls an internal object routine to delete the principal ID. This routine is identical in function to `obj$remove_obj_from_container` with the following exception: The object ID count is examined once all the mutexes have been acquired. If the object ID count is not one, the principal ID is not deleted.
   - Returns to caller.

10. If the resultant object ID count is greater than one, or the object ID count is one and the source object is either not temporary or its principle object ID has already been deleted, then `obj$remove_obj_from_container` performs the following steps:
   - Removes the object ID and name from the source container.
   - Releases the source container directory level, the source container, and the type-specific mutexes.
   - Return to caller.

The procedure `obj$dereference_object` performs the following steps:

1. Decrement the pointer count field in the object header using the RMALI instruction.

2. If the pointer count field becomes zero, `obj$dereference_object` performs the following steps:
   - Issues a bug check if the object ID field is not zero.
   - Calls the OTD delete routine to deallocate the object body extensions and any associated structures.
   - Unlinks the object header from the OTD list.
   - Decrements the create count field in the OTD.
   - Deletes the object header and object body storage allocated in the `obj$create_initialize_object` object call.

3. Returns to the caller.
There is one interesting race condition that can arise during the deletion of an object ID that refers to a temporary object. This race condition, however, does not cause any system database to be corrupted. Assume that there are two threads that are simultaneously executing on two different processors: one is creating a reference to a temporary object, and the other is deleting a reference to the same temporary object. Further assume that the reference being deleted is the only reference to the object save its principle object ID.

Both threads execute the locking sequence using two different object IDs, and thus possibly acquiring different mutexes for the container directory and object container. The type-specific mutex, however, is the same, and one or the other of the threads acquires the mutex first. If the thread that acquires the mutex is the thread creating a reference, then nothing unusual can occur. If the thread that is deleting a reference acquires the mutex first, then the following anomaly can occur.

The thread deleting the reference discovers that the resultant object ID count is one, the subject object is temporary, and that its principle object ID has not been deleted. The reference object ID is deleted, the mutexes are released, and the Delete Object ID routine is called, specifying the principle object ID. But before the principle object ID can be deleted, the other thread completes the creation of the reference, thus incrementing the object ID count back to 2. It then releases its mutexes, whereupon the original thread now acquires the appropriate mutexes and checks the object ID count. The object ID count is not one and the principle object ID of the object is not deleted.

5.7.8 Transferring an Object Container

When a new process is created, it is given two object containers to serve as its display- and private-process level object containers. These object containers are created as part of process creation. See Chapter 6, Process Structure for more details on the Create Process and Job system service.

The object architecture provides support for the process and job creation services to transfer object containers from one job/process to a new job/process.

When an attempt is made to transfer an object container, every object in the container must meet the following conditions:

1. The thread issuing the system service must have write access to the object container.
2. The object being transferred must have an object ID count of one. This also prevents temporary objects from being transferred.
3. Reference objects must refer to a system level object.
4. The object being transferred must have the transfer action flag in the TRANSFER state.
5. If the object is allocated, the thread doing the transfer must pass the allocation check.

When an object is transferred, its name, if any, is transferred as well. If the object was allocated at the job, process, or thread level, then its allocation is transferred to the new entity being created. For example, if a job is being created, the object's allocation would transfer to the job level.

When a container is transferred, Mica performs the following steps:

1. Acquires the directory mutex.
2. Translates the container ID to an object header pointer with access as write.
3. Ensures that the resulting container object transferable. An error condition is raised if the container is not transferable. Displayed containers or the default private container are set as no transfer.
4. For each element in the object container Mica performs the following steps:
   - Checks to see if the object ID is a reference ID.
   - Checks the allocation of the object if the object ID is not a reference ID. If the allocation is at at thread level, then Mica transfers the allocation to the new entity.
If the object ID is a reference ID, checks to ensure that the principal ID is at the system level.

If all objects satisfy the criteria, removes the object container from the container directory and release the directory mutex.

5. Adds a pointer to the object container in the new container directory.

6. Updates the object header to point to the new container directory, also, the ownership of the objects needs to be changed to match the owner of the new entity.

5.7.9 Create Reference

When an object has a pointer to it outstanding, its data structures cannot be deleted. The executive can increment the pointer count field in the object header to prevent an object's storage from being deleted. A user-mode program may achieve similar behavior by creating a reference ID to the object ID.

Creating a reference ID creates another object ID for an object. When a reference ID is created for an object, the object ID count in the object's header is incremented. This allows a user to create a reference ID to an object, and thus ensures that the object's storage cannot be deleted until the reference object ID is deleted.

The level of the target container that receives the new reference ID must be less visible than the container of the principal object ID.

The exec$create_reference_id service, and the exec$make_temporary service both create reference IDs to the specified object.

The declaration for this executive function is:

```pascal
PROCEDURE exec$create_reference_id ( 
    IN mode : INTEGER; 
    IN source_object_body : exec$object_id; 
    IN target_container : exec$object_id; 
) RETURNS reference_object_id;
```

The exec$create_reference_id routine performs the following steps:

1. Raises an error condition if the level of the target_container is not less visible than the level of the source container.

2. Acquires the target_container directory-level mutex.

3. Acquires the target_container mutex.

4. Acquires the source container directory-level mutex.

5. Translates the source object ID to a pointer.

6. Checks for read access.

7. Checks the allocation class of source object container.

8. If the source object is of type container, then exec$create_reference_id releases the target container directory level, target container, and source container directory-level mutexes. An error condition is raised.

9. Allocates the object index in the target container.

10. If allocation of the object index fails, then exec$create_reference_id releases the target container directory level, target container, and source container directory-level mutexes. An error condition is raised.

11. Acquires the source object type-specific mutex.

5–38 Object Architecture
12. Increments the object ID count of the source object.

13. Stores a pointer to the source object in target container at allocated index position.

14. Constructs the object ID of reference using the target container directory, the allocated target container index, and the sequence number of the source object.

15. Releases the target container directory level, target container, source container directory level, and type-specific mutexes.

16. Returns the object ID of newly created reference.

5.7.10 Make Temporary

Making an object temporary creates a new principal object ID for the object at a more visible level and marks the object as temporary. The original object ID becomes a reference object ID. If the object has a name, then the name is moved to the target object container.

When an object is made temporary, it is possible that a name conflict exists in the target container. The caller can specify that the object that is being made temporary is to be deleted and its object ID is to become a reference to the object which caused the name conflict. This provides the capability for two or more processes to share objects without regard to who created them first.

The declaration for this executive function is:

```plaintext
PROCEDURE e$make_temporary (  
   IN mode : INTEGER;  
   IN replacement : BOOLEAN;  
   IN source : exec$object_id;  
   IN target_container : exec$object_id;  
) RETURNS reference_id : exec$object_id;
```

The `e$make_temporary` routine performs the following steps to make an object temporary:

1. If the level of the target container is not more visible than the source container, then `e$make_temporary` raises an error condition.

2. Acquires source container directory-level mutex.

3. Translates source object ID to a pointer.

4. Releases the source container directory-level mutex and raises an error condition if the source object is of type container.

5. Checks for delete access on source object.

6. Acquires target container directory-level mutex.

7. Acquires target container mutex.

8. Checks for temporary create access on target object container.


10. If the source object ID is not the principle object ID, the source object ID count is not 1, or the source object is already temporary, then `e$make_temporary` releases the source container directory level, the target container directory, the target container, and the type-specific mutexes. An error condition is raised.

11. Checks for name conflict in target container if source object has a name.

12. If a name conflict exists and replacement is not specified, then `e$make_temporary` releases the source container directory level, the target container directory, the target container, and type-specific mutexes. An error condition is raised.
13. If a name conflict exists and replacement is specified, then e$make_temporary performs the following steps:
   - Raises an error condition if the object whose name conflicts is not temporary.
   - Increments the object ID count of the object in the target container whose name conflicts with the source object and save pointer to object.
   - Releases the target container directory level and target container mutexes.
   - Decrements the object ID count of the source object (the result is guaranteed to be zero).
   - Releases the type-specific mutex.
   - Calls the type-specific remove object ID routine for the source object.
   - Deletes the name of source object from source container.
   - Stores a pointer to the target container object in source container at the source object ID.
   - Retrieves the sequence number from the target object, and inserts in the source object ID.
   - Releases the source container directory-level mutex.
   - Returns the source object ID with the new sequence number inserted.


15. If allocation of the object index fails, then e$make_temporary releases the source container directory-level, the target container directory, the target container, and type-specific mutexes. An error condition is raised.

16. Stores a pointer to the source object in target container at the allocated index position.

17. Removes the source object name, if any, from its previous name table, and inserts it in the new target container name table.

18. Sets the principle object container address to the target container address.

19. Marks the source object temporary and indicates that a make temporary operation was performed.

20. Increments the object ID count of source object.

21. Releases the source container directory-level, the target container directory, the target container, and the type-specific mutexes.

22. Returns the source object ID.

5.7.11 Mark Temporary

The Mark Temporary routine allows an object to be marked as temporary after it has been created. This operation is used to cause a permanent object to be deleted when all its reference object IDs are deleted. If the object has no reference object IDs, then its principle object ID is deleted immediately. Objects can only be marked temporary that reside at the job or system levels.

The declaration for this executive procedure is:

```
PROCEDURE e$mark_temporary (    
     IN mode : INTEGER;    
     IN source : exec$object_id;    
 )
```

The e$mark_temporary routine performs the following steps.

1. Raises an error condition if the level of the source object is a process.
2. Acquires the source container directory-level mutex.
3. Acquires the source container mutex.
4. Translates the source object ID to a pointer.
5. Checks for delete access on source object.
6. Acquires the source object type-specific mutex.
7. Sets the temporary flag in the source object and indicates that a mark temporary was performed. Note, if the object was already temporary, it does not change the state of the objhdr$temporary_operation flag.
8. If object ID count of source object is 1 and the source object ID is the principle object ID, then e$mark_temporary performs the following steps:
   • Decrements the object ID count (it is guaranteed to be zero).
   • Releases the type-specific mutex.
   • Calls the type-specific remove object ID routine.
   • Deletes the object ID and name from source container.
   • Releases the source container directory-level and source container mutexes.
   • Returns to the caller.
9. If object ID count of source object is not 1 or the source object ID is not the principle object ID, then e$mark_temporary releases the source container directory level, the source container, and the type-specific mutexes.
10. Returns to the caller.

5.8 Allocating an Object
An object may be allocated to one of the following classes:
• Identifier object
• User object (This is the user object which exists for each active user on the system.)
• Job object
• Process object
• Thread object

The identifier object class consists of all the identifiers in the rights database which have corresponding identifier allocation objects. Identifier allocation objects are created with the following service and exist in the system level container mica$identifier_allocation.

PROCEDURE exec$create_identifier_allocation (  
   OUT identifier_allocation_id : exec$object_id;  
   IN object_item_list : exec$item_list;  
   IN identifier : INTEGER;  
) RETURNS STATUS;

The active user ID class consists of the set of all user IDs currently active on the system. An object which is allocated to an active user ID is automatically deallocated when no users of that ID are currently on the system. For example, if an object is allocated to user KOLSEN, it is automatically deallocated when the last user with the ID KOLSEN is removed from the system.

The job class consists of the set of all threads within the job. When the job terminates, all objects allocated to the job class are automatically deallocated.

Object Architecture 5-41
The process class consists of the set of threads within the specified process, and any thread within a descendant process. When the specified process terminates, all objects allocated to that process class are automatically deallocated.

The thread class consists of the thread that allocated the object. When the thread terminates, all objects allocated to the thread class are automatically deallocated.

The $allocate_object$ argument that specifies the visibility of the object ID determines the allocation classes to which an object can be allocated. The rules for determining allocation class are as follows:

- If the ID is at the system level, the object may be allocated to any one of the five classes.
- If the ID is at the job level, the object may only be allocated to the job, process, or thread classes.
- If the ID is at the process level, the object may only be allocated to the process or thread classes.

5.8.1 Object Allocation Block

When an object is allocated the object header contains a pointer to the object allocation block. The object allocation block contains:

- A forward link to the next allocation block in this class.
- A backward link to the previous allocation block in this class.
- The allocation type (identifier, active user, job, process or thread).
- The allocation ID which identifies the allocation. For example, if the allocation is identifier, the ID of the identifier object is stored here.
- A pointer to the object header which refers to this allocation block.

Each allocation class has a listhead. The listhead for the thread allocation class is located in the thread object, the process in the process object, the job in the job object. The listhead for each active user is maintained in the "user" object structure. Job, process, thread and user objects are defined in Chapter 6, Process Structure. The listhead for an identifier object is located in the specific identifier object.

The listhead offset is stored in the OTD for the object type. This allows the object architecture to deallocate all object allocated to a particular object (like a thread) when that object is deleted.

There is a single mutex which guards access to all allocation class listheads. This mutex is known as the allocation mutex.

The $allocate_object$ routine allocates the specified object to the specified allocation object. The declaration of this procedure is:

```
PROCEDURE e$allocate_object (  
   IN object_id : exec$object_id;  
   IN allocation_id : exec$object_id;  
   IN access_mode : integer;  
);  
```

The $allocate_object$ routine performs the following:

1. Translates the allocation_id with an access type of read to obtain the object class and other information.
2. Ensures that the calling thread is within the specified allocation class.
3. Ensures that the allocation class is compatible with the visibility of the specified object ID.

5-42 Object Architecture
4. Translates the specified ID by calling obj$reference_object_by_id, with an access type of ALLOCATE.

5. Acquires an allocation mutex if the translation succeeds.
   This prevents other threads from attempting to allocate the same object simultaneously.

6. Checks to ensure that the object ID count for the object is 1.

7. Checks to ensure that the object is not currently allocated.

8. If all checks pass, e$allocate_object calls the OTD allocate routine for that object type, passing in the object body and the allocation type.

9. If the allocate routine does not return denial, e$allocate_object allocates an object allocation block, initializes the block, and links it into the appropriate allocation list.

10. Adds a pointer to the object allocation block to the object header and from the object header to the allocation block.

11. Releases the allocation mutex.

12. Dereferences the allocation object.

13. Dereferences the allocated object.

14. Returns status to the user.

We need to be able to have a protected subsystem assume the allocation class as the thread whose behalf it is working

5.9 Deallocating an Object

Allocated objects may be explicitly deallocated via a call to exec$deallocate_object or implicitly deallocated when an allocation class ceases to exist. For example, when a process terminates, any objects allocated to that process are implicitly deallocated.

The e$deallocate_object procedure deallocates the specified object. The declaration of this procedure is:

```fortran
PROCEDURE e$deallocate_object (  
   IN object_id : exec$object_id;  
) RETURNS STATUS;
```

The e$deallocate_object routine performs the following:

1. Translates the specified object ID by calling obj$reference_object_by_id with an access type of DEALLOCATE.

2. If obj$reference_object_by_id succeeds and the object was allocated, then the thread is in the allocation class of the object.

3. Acquires the allocation mutex.

4. Checks to see if the object is currently allocated, if the object is not allocated, release the allocation mutex, call obj$dereference_object and return an error status of not allocated.

5. Calls the type-specific deallocate routine, passing in the address of the object body and the allocation type.

6. Clears the allocation pointer in the object's header.

7. Unlinks the object allocation block, and calls obj$dereference_object.

8. Releases the allocation mutex.

9. Returns the storage occupied by the object allocation block to the system.
10. Returns status to the user.

Implicit deallocation which occurs when an allocation class terminates. For example, upon thread termination, $deallocate_object does the following:

1. Acquires the allocation mutex.
2. Removes the first entry from the terminating class's allocation list.
3. Uses the back pointer to the object header get the object type.
4. Calls the type-specific deallocate routine passing in the address of the object header and the allocation type.
5. Clears the allocation pointer in the object's header.
6. Unlinks the object allocation block.
7. Releases the allocation mutex.
8. Returns the storage occupied by the object allocation block to the system.
9. Repeats from step 1 until there are no more entries on the list.

5.10 Quotas and Objects

Objects are allocated from system paged and nonpaged pool, and as such are charged against the quota for a process or a job. The following rules indicate how and what is charged for object creation and reference. When an object is deleted, the same rules are used to return quota.

- Objects created in system level object containers are not charged against quotas. Objects at this level are permanent, and may outlive the life of the entity which created the object.
- Objects which are created at job or process level are charged to the level at which the object was created.
- If an object is permanent or was at one time permanent, (that is, marked as temporary) there are no charges for references. Note, objects of this type are in job or system containers.
- If an object was made temporary, a charge is made for the reference at the level of the reference. Since the creator of the object did not desire to create a permanent object, each reference of the object is charged the total cost of the object. Note, the entity that initially created the object was charged for the object creation, and is not charged again when the object is made temporary.

The exact methodology of charging and releasing quotas is still under discussion. One method is to add a field to the object header that indicates the charged amount for this object. This charged amount field would be used for creation, references, transfers, and deletion.

Chapter 6, Process Structure should describe a pair of procedures to charge and return quotas that allow the user to specify the amount, the type, and the entity to charge.

5.11 Executive Support Functionality

This section describes the executive procedures which are invoked by object service routines for standard operations. Using a single set of executive procedures for common object operations enhances reliability and maintainability.
5.11.1 obj$reference_object_by_id

Given an object ID, type, mode and desired access, this procedure returns the address of the object body for the ID and increments the pointer count field for the object. An error is returned if the object ID is invalid.

PROCEDURE obj$reference_object_by_id (  
  IN object_id : e$object_id;  
  IN object_type : e$object_type;  
  IN access_mode : k$processor_mode;  
  IN desired_access : e$access_type;  
  OUT object_body : POINTER anytype CONFORM;  
) RETURNS STATUS;

5.11.2 obj$translate_object_name

Given an object container, object name, object type and access mode, this procedure returns the object ID corresponding to the name.

PROCEDURE obj$translate_object_name (  
  IN container_id : e$object_id;  
  IN name : string (*);  
  IN object_type : e$object_type;  
  IN access_mode : k$processor_mode;  
  IN case_blind : boolean;  
  OUT object_id : e$object_id;  
) RETURNS STATUS;

5.11.3 obj$create_initialize_object

This routine is called with the number of bytes to allocate in addition to the object header. It allocates the storage from the appropriate pool, and initializes the object header.

PROCEDURE obj$create_initialize_object (  
  IN otd_id : e$object_id;  
  IN access_mode : k$processor_mode;  
  IN acl : POINTER e$access_control_list;  
  IN name : string (*);  
  IN transfer : boolean;  
  IN reference_inhibit : boolean;  
  IN pool_type : e$pool_index;  
  IN object_size : integer;  
  OUT object_body : POINTER anytype CONFORM;  
) RETURNS STATUS;

5.11.4 obj$insert_object_in_container

This routine accepts an object body pointer and a object container ID, and adds the object to the specified object container returning the new object ID.

PROCEDURE obj$insert_object_in_container (  
  IN object_body : POINTER anytype CONFORM;  
  IN access_mode : k$processor_mode;  
  IN container_id : e$object_id;  
  OUT object_id : e$object_id;  
) RETURNS STATUS;
5.11.5 obj$insert_object_and_reference

This routine accepts an object body pointer and a object container ID, and adds the object to the specified object container returning the new object ID and pointer to the corresponding object body. The reference count on the object is increased by one.

This procedure is provided for object services which need to immediately reference an object after it has been created. It avoids race conditions with the object name colliding, the ID of the colliding object being returned, and before the object can be referenced, the object is deleted.

```plaintext
PROCEDURE obj$insert_object_and_reference (  
    IN object_body : POINTER anytype CONFORM;  
    IN access_mode : k$processor_mode;  
    IN container_id : e$object_id;  
    OUT object_id : e$object_id;  
    OUT new_object_body : POINTER anytype CONFORM;  
) RETURNS STATUS;
```

5.11.6 obj$remove_obj_from_container

This routine accepts an object ID and removes the object ID from its associated object container.

```plaintext
PROCEDURE obj$remove_obj_from_container (  
    IN object_id : exec$object_id;  
    IN access_mode : k$processor_mode;  
) RETURNS STATUS;
```

5.11.7 obj$dereference_object

This routine decrements the pointer count of the specified object. If the resulting pointer count is zero, the object deletion routine for the specified object is invoked.

```plaintext
PROCEDURE obj$dereference_object (  
    IN object_body : POINTER anytype CONFORM;  
);
```

5.11.8 obj$get_principal_object_id

This routine returns the principal object ID for a given object body. If the object has no principal ID, the invalid ID zero is returned.

```plaintext
PROCEDURE obj$get_principal_object_id (  
    IN object_body : POINTER anytype CONFORM;  
) RETURNS e$object_id;
```

5.11.9 obj$set_object_acl

This procedure updates the ACL field in the object header.

```plaintext
PROCEDURE obj$set_object_acl (  
    IN object_body : pointer exec$object_body;  
    IN acl : pointer exec$sacl;  
) RETURNS old_acl POINTER exec$sacl;
```

5-46 Object Architecture
CHAPTER 6
PROCESS STRUCTURE

6.1 Introduction
This chapter describes the external interfaces and data structures of the Mica process structure, the architecture of which is based on the User, Job, Process, Thread (UJPT) hierarchy. This chapter also describes the UJPT implementation in terms of its algorithms and dependencies on other portions of the Mica system (e.g. the kernel and object architecture).

6.2 Goals/Requirements
The goal of the UJPT architecture is to provide a vehicle for controlling multiple threads of execution in a single address space. The architecture provides facilities for resource usage control, security profile management, address space and image management, and object container directory services.

6.3 UJPT Hierarchy
The UJPT architecture consists of a hierarchy of objects. The objects provide a logical grouping of functionality and control.

6.3.1 The User Object
The User object appears at the highest level of the UJPT hierarchy. Its primary function is to provide a focal point for acquiring security profiles and resource quotas/limits for its underlying objects.

The User object is implemented as a system level object in the "USER$OBJECT_CONTAINER" object container.

6.3.1.1 Object Structure
Each user of the Mica system is assigned a unique username, a security profile, and a set of resource limits or quotas. The Mica system keeps track of this information in a system-wide authorization file. If the user has at least one active job, the information is also kept in his user object. As we shall see later in this chapter, information from the user object is propagated down the UJPT hierarchy on an as-needed basis.

NOTE
The intent of the Mica executive is to remain independent of the system-wide authorization file. Therefore, all Mica user attributes are stored in the user object. In addition, the Mica executive places no restrictions on the source of information stored in the user object. It does, however, place a Digital-reserved identifier in the ACL for the user object OTD which limits who can create user objects.
The user object is split into a user object body and a user control block. The user object body contains the information necessary to support the UJPT hierarchy. The user control block contains the vital information of the user object. Example 6–1 illustrates the data structures used to represent the user object.

**Example 6–1: User Object Structure**

```plaintext
e$User_object_body: RECORD
  u_obj_id: e$object_id;
  u_user_flags: e$User_flags;
  u_job_queue_mutex: k$mutex;
  u_job_count: integer;
  u_job_queue_hd: e$linkedList;
  u_uob: e$User_control_block;
END RECORD;

e$User_control_block: RECORD
  ucb_username: string(e$MAX_USER_NAME);
  ucb_security_profile: e$security_profile;
  ucb_quotas: e$quotas;
  ucb_thread_priority: k$combined_priority;
  ucb_access_restrictions: e$access_restrictions;
  ucb_user_allocation_list: e$allocation_list;
  ucb_thread_priority: k$combined_priority;
END RECORD;
```

### 6.3.1.1 Security Profile

The security profile maintained in the user object contains the list of identifiers assigned to the Mica user. The identifier list gives access rights to the user object as described in Chapter 10, Security and Privileges.

### 6.3.1.2 Resource Control

The goals of the Mica system resource control and quota architecture are:

- Prevent a single user from abusing the system by over running system resources.
- Be simple, predictable and easy to understand.
- Provide repeatable consistent behavior.

The Mica system achieves these goals through data structures maintained in the user object and through policies implemented in the object architecture, memory management system, and the kernel. Example 6–2 illustrates the resource-control data structures maintained in the user object.
Example 6–2: Resource Control Structures

| User Object Resource Control |

  e$quotas: RECORD
  q_usage_and_limits: e$quota_usage_and_limits;
  q_per_job_limits: e$quota_limits;
  q_per_process_limits: e$quota_limits;
  END RECORD;

  e$quota_level : (  
    e$g_user_quota_level,
    e$g_job_quota_level,
    e$g_process_quota_level
  );

  e$quota_types : (  
    e$g_paging_file_quota,
    e$g_paged_pool_quota,
    e$g_nonpaged_pool_quota,
    e$g_cpu_time_quota
  );

  e$quota_vector: ARRAY[e$quota_types] OF e$counter;

  ! Resource Limits

  e$quota_limits: e$quota_vector;

  ! Resource Usage

  e$quota_usage: e$quota_vector;

  ! Quota Usage and Limits

  e$quota_usage_and_limits: RECORD
  qual_mutex: k$mutex;
  qual_limits: e$quota_limits;
  qual_usage: e$quota_usage;
  END RECORD;

During user-object creation, the ucb_quotas field of the user control block is initialized. The values are obtained from the user_record parameter to the exec$create_user() system service.

Once established, the ucb_quotas field of the user control block becomes the focal point for resource allocation limitation. Resources are allocated in bulk from the user-object to the job-object, and from the job-object to the process object at object creation time. The allocation amount is specified by the q_per_job_limits, or q_per_process_limits in the user-object. The allocation is regulated by the amount of quota currently in use by the object that the allocation is coming from.

6.3.1.1.3 Access Restrictions

The user object maintains the current system access restrictions for the Mica user that it represents. The access restrictions are not enforced by the UJPT architecture. External processes may inspect the access restrictions in the current set of user objects and determine what type of enforcement actions are necessary. Example 6–3 illustrates the data structures used to maintain the access restrictions placed in the user object.
Example 6-3: Access Restriction Data Structures

| e$access_restrictions : RECORD
|   ar_restriction_vector : ARRAY[e$job_class] OF e$class_access_restrictions;
|   ar_expiration_date : e$date;
|   ar expiration_vector : ARRAY[7] OF e$date;
| END RECORD;

| e$class_access_restrictions : RECORD
|   car_prime_days : e$day_set;
|   car_non_prime_days : e$day_set;
|   car_prime_hours : e$hour_set;
|   car_non_prime_hours : e$hour_set;
| END RECORD;

6.3.1.2 Functional Interface

The Mica executive provides entry points capable of creating and deleting user objects, and setting and extracting various attributes of a User object.

6.3.1.2.1 User Creation

Creating a user object also causes a UPT hierarchy to be created. The system service exec$create_user() creates a user object, job object, process object, and thread object. If there is a name collision between the new user object and an existing user object for the same user, then the new user object is discarded, and the job, process, and thread objects are attached to the existing user object. Example 6-4 illustrates the interface to exec$create_user().

Example 6-4: User Object Creation System Interface

PROCEDURE exec$create_user (  
  OUT object_id : exec$object_id;
  IN object_parameters : exec$object_parameters = DEFAULT;
  IN user_record : exec$user_record;
  IN user_allocation_list : POINTER exec$allocation_list = NIL;
  IN job_object_parameters : exec$object_parameters = DEFAULT;
  IN job_record : exec$job_record = DEFAULT;
  IN job_initial_container : exec$object_id = DEFAULT;
  IN job_allocation_list : POINTER exec$allocation_list = NIL;
  IN process_object_parameters : exec$object_parameters = DEFAULT;
  IN process_record : exec$process_record;
  IN process_public_container : exec$object_id = DEFAULT;
  IN process_private_container : exec$object_id = DEFAULT;
  IN process_allocation_list : POINTER exec$allocation_list = NIL;
  IN process_data_block : POINTER quadword_data(*) CONFORM = NIL;

Example 6-4 Cont'd. on next page
Example 6-4 (Cont.): User Object Creation System Interface

```c
IN thread_object_parameters : execObject_parameters = DEFAULT;
IN thread_record : execThread_record = DEFAULT;
IN thread_allocation_list : POINTER execAllocation_list = NIL;
IN thread_data_block : POINTER quadword_data(*) CONFORM = NIL;
IN thread_immediate_parameters1 : execThread_parameter = 0;
IN thread_immediate_parameters2 : execThread_parameter = 0;
IN thread_status : execObject_id = DEFAULT;
) RETURNS status;

EXTERNAL;

++

Routine description:

Create a user, job, process, and thread object as specified by the parameters.
If the user object collides with an existing user object, then use the existing
user object

Arguments:

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>object_id</td>
<td>Object ID of the resulting user object</td>
</tr>
<tr>
<td>object_parameters</td>
<td>The object type independent parameters for the user object</td>
</tr>
<tr>
<td>user_record</td>
<td>The object type independent parameters for the user object</td>
</tr>
<tr>
<td>user_allocation_list</td>
<td>Attributes of new user (from authorization file ?)</td>
</tr>
<tr>
<td>job_object_parameters</td>
<td>Objects to be allocated to the user object. If not present then no objects are allocated to the user</td>
</tr>
<tr>
<td>job_record</td>
<td>The object type independent parameters for the job object</td>
</tr>
<tr>
<td>job_initial_container</td>
<td>Attributes of the job being created. If not present, then values are obtained from current user object</td>
</tr>
<tr>
<td>job_allocation_list</td>
<td>Job level object container to be transferred into the job</td>
</tr>
<tr>
<td>process_object_parameters</td>
<td>Job level container directory for this job. If not present then container directory comes up empty</td>
</tr>
<tr>
<td>process_record</td>
<td>Objects to be allocated to the job object. If not present then no objects are allocated to the job</td>
</tr>
<tr>
<td>process_public_container</td>
<td>The object type independent parameters for the process object</td>
</tr>
<tr>
<td>process_private_container</td>
<td>Attributes of the process being created</td>
</tr>
<tr>
<td>process_allocation_list</td>
<td>Process level public container to be transferred into the process</td>
</tr>
<tr>
<td>process_level_container</td>
<td>Process level container directory for the process. If not present then container comes up empty,</td>
</tr>
<tr>
<td>process_allocation_list</td>
<td>Process level private container to be transferred into the process</td>
</tr>
<tr>
<td>process_data_block</td>
<td>Process level container directory for the process. If not present then container comes up empty,</td>
</tr>
<tr>
<td>thread_object_parameters</td>
<td>Objects to be allocated to the process object. If not present then no objects are allocated to the process</td>
</tr>
<tr>
<td>thread_record</td>
<td>The object type independent parameters for the thread object</td>
</tr>
<tr>
<td>thread_allocation_list</td>
<td>Attributes of the thread being created</td>
</tr>
<tr>
<td>thread_data_block</td>
<td>Objects to be allocated to the thread object. If not present then no objects are allocated to the thread</td>
</tr>
<tr>
<td>thread_immediate_parameters1</td>
<td>Arbitrary data block passed to the process</td>
</tr>
<tr>
<td>thread_immediate_parameters2</td>
<td>Arbitrary data block passed to initial thread. Pointer in TCR, if pointer is NIL, then no data block was passed</td>
</tr>
<tr>
<td>thread_status</td>
<td>Immediate parameter passed to thread through TCR</td>
</tr>
<tr>
<td></td>
<td>Immediate parameter passed to thread through TCR</td>
</tr>
<tr>
<td></td>
<td>Exit status object to be bound to the initial thread. If not present then the thread is created without an exit status object</td>
</tr>
</tbody>
</table>

Return value:

TBS

-
From the interface to exec$create_user(), it is clear that the user_record can have an impact on the structure of the user being created. Example 6–5 illustrates the layout of the user_record.

Example 6–5: User Record Structure

```
! The User Record
!
exec$user_record : RECORD

! User Fields
!
! The User fields are only used to initialize a user object if no user
! object exists. The intent is for the contents of these fields come from
! the system wide authorization file

user_username : string(e$cm_user_name);
user_security_profile : e$security_profile;
user_per_user_limits : e$quota_limits;
user_per_job_limits : e$quota_limits;
user_per_process_limits : e$quota_limits;
user_thread_priority : k$combined_priority;
user_thread_affinity : k$affinity;
user_access_restrictions : e$access_restrictions
END RECORD;
```

6.3.1.2.2 Get/Set User Information

The exec$get_user_information and exec$set_user_information system services provide a mechanism to obtain and to modify attributes of the specified user object. Example 6–6 illustrates the interfaces to the user object get/set system services.

Example 6–6: Get/Set User Information System Interface

```
PROCEDURE exec$get_user_information (
    IN user_object_id : exec$object_id = DEFAULT;
    IN user_get_items : POINTER exec$item_list_type;
) RETURNS status;
EXTERNAL;

+++ 
| ! Routine description:
| ! Return information about the user object to the caller. The
| ! information returned is item list driven
| Arguments:
| user_object_id if present, the object ID of user object that is to be inspected
| otherwise, the user object of the calling thread is assumed
| user_get_items item list identifying user object information to be extracted
|!
| Return value:
| TBS
| --

PROCEDURE exec$set_user_information (
    IN user_object_id : exec$object_id = DEFAULT;
    IN user_get_items : POINTER exec$item_list_type;
) RETURNS status;
EXTERNAL;
```

Example 6–6 Cont’d. on next page
Example 6–6 (Cont.): Get/Set User Information System Interface

```c

| Arguments: |
| user_object_id | if present, the object ID of user object that is to be modified |
| user_get_items | item list identifying user object information to be modified |

| Return value: |
| TBS |

```

Only certain pieces of the user object may be inspected or modified. Table 6–1 illustrates the possible item codes and the information read or written when using the item code.

<table>
<thead>
<tr>
<th>Item Code</th>
<th>Set Action</th>
<th>Get Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>e$C_job_count</td>
<td>error</td>
<td>return u_job_count</td>
</tr>
<tr>
<td>e$C_job_ids</td>
<td>error</td>
<td>return object ID's of jobs owned by user</td>
</tr>
<tr>
<td>e$C_username</td>
<td>error</td>
<td>return username of user</td>
</tr>
<tr>
<td>e$C_security_profile</td>
<td>replace ucb_security_profile</td>
<td>return ucb_security_profile</td>
</tr>
<tr>
<td>e$C_quotas</td>
<td>error</td>
<td>return ucb_quotas</td>
</tr>
<tr>
<td>e$C_user_limits</td>
<td>replace qual_limits</td>
<td>return qual_limits</td>
</tr>
<tr>
<td>e$C_job_limits</td>
<td>replace a_per_job_limits</td>
<td>return a_per_job_limits</td>
</tr>
<tr>
<td>e$C_process_limits</td>
<td>replace a_per_process_limits</td>
<td>return a_per_process_limits</td>
</tr>
<tr>
<td>e$C_thread_priority</td>
<td>replace ucb_thread_priority</td>
<td>return ucb_thread_priority</td>
</tr>
<tr>
<td>e$C_access_restrictions</td>
<td>replace ucb_access_restrictions</td>
<td>return ucb_access_restrictions</td>
</tr>
<tr>
<td>e$C_allocation_list</td>
<td>error</td>
<td>return ucb_user_allocation_list</td>
</tr>
</tbody>
</table>

6.3.1.2.3 User Deletion

The `exec$/force_exit_user()` system service provides a mechanism for removing an active Mica user from the system. The service effectively causes an entire UJPT hierarchy to be removed, including all jobs, processes, and threads that are directly beneath the user object. Example 6–7 illustrates the interface used to remove a user object from the Mica system.
Example 6-7: User Object Deletion System Interface

PROCEDURE exec$force_exit_user (  
    IN user_object_id : exec$object_id = DEFAULT;  
    IN exit_status : status;  
) RETURNS status;  
EXTERNAL;

+++  
Routine description:  
Causes the UJPT hierarchy whose user object head is user_object_id to  
be removed from the Mica system  

Arguments:  
user_object_id  
  the user object to be removed. If not specified,  
    then the current user is assumed  
extit_status  
  the reason that the user is force-exiting  

Return value:  
TBS  
+-

6.3.2 The Job Object

The job object appears at the second level of the UJPT hierarchy. Its sole function is to provide a  
set of resource limits for a collection of processes running as a job. The job object also provides a job  
level container directory.

The job object is implemented as a system level object in the "JOB$OBJECT_CONTAINER" object  
container.

6.3.2.1 Object Structure

Each job in the Mica system represents a set of active processes and is responsible for controlling the  
resources used by those processes.

The job object is split into a job object body and a job control block. The job object body contains the  
information necessary to maintain its position in a UJPT hierarchy. The job control block contains  
the information necessary to provide resource management for the job's processes. Example 6-8  
illustrates the job object.  

6-8 Process Structure
Example 6-8: Job Object Structure

```
job_object_body : RECORD
  j_obj_id : eobject_id;                  ! Object ID of the job object
  j_user_pointer : POINTER eUser_object_body;  ! Referenced Pointer to owning User
  j_job_flags : ejob_flags;            ! Job Flags
  j_job_queue : elinked_list;          ! List of users jobs
  j_process_count : integer;           ! Mutex for process management
  j_process_queue_hd : elinked_list;   ! Number Of processes of the job
  j_job : ejob_control_block;          ! List head of jobs processes
END RECORD;

job_control_block : RECORD
  j_job_class : ejob_class;               ! Job Control Block
  j_job_usage_and_limits : ejob_usage_and_limits;    ! The jobs class
  j_job_jobcondir_mutex : kmutex;           ! Current resources used/resource limits
  j_job_condir_mutex : kmutex;             ! Job Level Condir mutex
  j_job_condir_id : eobject_id;            ! Process Level Condir mutex
  j_job_condir_leaf : eobject_header;      ! Job Level Container directory ID
  j_job_condir_ptr : POINTER eobject_header; ! visible in jobs context
  j_job_initial_object : eobject_id;       ! Job Level Container directory ID
  j_job_allocation_list : eallocation_list; ! visible in an arbitrary context
  j_job_condir_list : eobject_header;      ! Pointer to Job Level Condir
  j_object_id : eobject_id;               ! Object ID of the job initial container
  j_objects_allocated_to : eobject_list;   ! Objects allocated to the job objects
END RECORD;
```

6.3.2.1.1 Resource Control

The job object maintains resource usage information for itself, in addition to providing a pool of resources to its processes on an as-needed basis. During job-object creation, the `job_usage_and_limits.qualification` field of the job control-block is set to the value of `q_per_job_limits` from the user control block. The `job_usage_and_limits.qualification` field of the job control-block is then set to zero(), and the `q_usage_and_limits.qualification` field of the user control block is incremented by `q_per_job_limits` to reflect the resources allocated to the job. Once this resource shuffling operation has completed, the value of `job_usage_and_limits.qualification` represents the amount of system resources available to the job object and to all its process.

While the above resource allocation scheme is the normal case, during job creation a parameter specifying the per-job limits for the job can be specified, altering the algorithm. This value simply overrides the value from `q_per_job_limits` in the above example and applies to the newly created job.

6.3.2.2 Functional Interface

The Mica executive provides entry points capable of creating and deleting job objects, and setting and extracting various attributes of a job object.

As part of job object creation, all of the necessary support data structures are created, including a job level container directory and associated kernel mutex dispatcher object.
6.3.2.2.1 Job Creation

The system service exec$create_job() causes the creation of a job object, a process object, and a thread object. These objects appear beneath the user object of the calling thread. Example 6-9 illustrates the interface to exec$create_job().

Example 6-9: Job Object Creation System Interface

```
PROCEDURE exec$create_job (  
    OUT object_id : exec$object_id;  
    IN object_parameters : exec$object_parameters = DEFAULT;  
    IN job_record : exec$job_record = DEFAULT;  
    IN job_initial_container : exec$object_id = DEFAULT;  
    IN job_allocation_list : POINTER exec$allocation_list = NIL;  
    IN process_object_parameters : exec$object_parameters = DEFAULT;  
    IN process_record : exec$process_record;  
    IN process_public_container : exec$object_id = DEFAULT;  
    IN process_private_container : exec$object_id = DEFAULT;  
    IN process_allocation_list : POINTER exec$allocation_list = NIL;  
    IN process_data_block : POINTER quadword_data(*) CONFORM = NIL;  
    IN thread_object_parameters : exec$object_parameters = DEFAULT;  
    IN thread_record : exec$thread_record = DEFAULT;  
    IN thread_allocation_list : POINTER exec$allocation_list = NIL;  
    IN thread_data_block : POINTER quadword_data(*) = NIL;  
    IN thread immediate parameter1 : exec$thread_parameter = 0;  
    IN thread immediate parameter2 : exec$thread_parameter = 0;  
    IN thread status : exec$object_id = DEFAULT;  
) RETURNS status;  
EXTERNAL;
```

++
| Routine description: |
| Create a job, process, and thread object as specified by the parameters. |
| If no user object exists, then also create a user object. |
| Arguments: |
| object_id | Object ID of the resulting job object |
| object_parameters | The object type independent parameters for the job object |
| job_record | Attributes of the job being created. If not present, then values are obtained from current user object |
| job_initial_container | Job level object container to be transferred into the job level container directory for this job. If not present then container directory comes up empty |
| job_allocation_list | Objects to be allocated to the job object. If not present then no objects are allocated to the job |
| process_object_parameters | The object type independent parameters for the process object |
| process_record | Attributes of the process being created |
| process_public_container | Process level public container to be transferred into the process level container directory for the process. If not present then container comes up empty |
| process_private_container | Process level private container to be transferred into the process level container directory for the process. If not present then container comes up empty |
| process_allocation_list | Objects to be allocated to the process object. If not present then no objects are allocated to the process |
| process_data_block | Arbitrary data block passed to the process |

Example 6-9 Cont’d. on next page
Example 6–9 (Cont.): Job Object Creation System Interface

```
: thread_object_parameters: The object type independent parameters for the thread object
: thread_record: Attributes of the thread being created
: thread_allocation_list: Objects to be allocated to the thread object. If not present then
: no objects are allocated to the thread
: thread_data_block: Arbitrary data block passed to initial thread. Pointer in TCR, if
: pointer is NIL, then no data block was passed
: thread_immediate_parameters1: Immediate parameter passed to thread through TCR
: thread_immediate_parameters2: Immediate parameter passed to thread through TCR
: thread_status: Exit status object to be bound to the initial thread. If not present
: then the thread is created without an exit status object

Return value:
TBS
```

From the interface to exec$create_job(), it is clear that the job_record can have an impact on the structure of the job being created. Example 6–10 illustrates the layout of the job_record.

Example 6–10: Job Record Structure

```

exec$job_record : RECORD

  : Job Fields

  job_class : es$job_class;            ! The class of the job being created (i.e. network, batch...)

  : Per Job Resource limits. This value is used as the
  : qual_limits value for the job object, and is deducted
  : from the qual_usage field of the owning user object.
  : A value of zero() in any one of fields means to use the
  : corresponding value of the q_per_job_limit from the
  : user structure

  job_per_job_limits : es$quota_limits;
END RECORD;
```

6.3.2.2.2 Job Deletion

The exec$force_exit_job() system service provides a mechanism for removing job objects from the system. The removal of a job has the following system-wide effects:

- All processes beneath the job are removed from the system.
- The amount of resources available to the job (qual_limits–qual_usage) is returned to the job's user object by decrementing qual_usage in the user object.
- If the job object is the last job owned by its user object, then the user object is removed from the system.

Example 6–11 illustrates the interface to exec$force_exit_job().
Example 6-11: Job Object Deletion System Interface

PROCEDURE exec$force_exit_job (  
   IN job_object_id : exec$object_id = DEFAULT;  
   IN exit_status : status;  
) RETURNS status;  
EXTERNAL;

++
| Routine description:
| Causes the job object specified by job_object_id to
| be removed from the Mica system
| Arguments:
| job_object_id the job object to be removed. If not specified,
| exit_status the reason that the job is force-exiting
| Return value:
| TBS
|--

6.3.2.2.3 Get/Set Job Information

The exec$get_job_information() and exec$set_job_information() system services provide a mechanism to obtain and to modify attributes of the specified job object. Example 6-12 illustrates the interfaces to the job object get/set system services.

Example 6-12: Get/Set Job Information System Interface

PROCEDURE exec$get_job_information (  
   IN job_object_id : exec$object_id = DEFAULT;  
   IN job_get_items : POINTER exec$item_list_type;  
) RETURNS status;  
EXTERNAL;

++
| Routine description:
| Return information about the job object to the caller. The
| information returned is item list driven
| Arguments:
| job_object_id if present, the object ID of job object that is to be inspected
| otherwise, the job object of the calling thread is assumed
| job_get_items item list identifying job object information to be extracted
| Return value:
| TBS
|--

PROCEDURE exec$set_job_information (  
   IN job_object_id : exec$object_id = DEFAULT;  
   IN job_get_items: POINTER exec$item_list_type;

Example 6-12 Cont'd. on next page

6-12 Process Structure
Example 6–12 (Cont.): Get/Set Job Information System Interface

```c
) RETURNS status;
EXTERNAL;

/**
 * Routine description:
 * Modify information in the job object. The
 * information to be modified is item list driven.
 *
 * Arguments:
 * job_object_id if present, the object ID of job object that is to be modified
 * otherwise, the job object of the calling thread is assumed
 * job_get_items item list identifying job object information to be modified
 *
 * Return value:
 * TBS
 */

Only certain pieces of the job object may be inspected or modified. Table 6–2 illustrates the possible item codes and the information read or written when using the item code.

<table>
<thead>
<tr>
<th>Item Code</th>
<th>Set Action</th>
<th>Get Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>e$C_user_id</td>
<td>error</td>
<td>return object ID of jobs user object</td>
</tr>
<tr>
<td>e$C_process_count</td>
<td>error</td>
<td>return j_process_count</td>
</tr>
<tr>
<td>e$C_process_ids</td>
<td>error</td>
<td>return object ID's of processes owned by job</td>
</tr>
<tr>
<td>e$C_usage_and_limits</td>
<td>error</td>
<td>return jop_usage_and_limits</td>
</tr>
<tr>
<td>e$C_job_limits</td>
<td>replace qual_limits</td>
<td>return qual_limits</td>
</tr>
<tr>
<td>e$C_job_condir_id</td>
<td>error</td>
<td>return jop_job_condir_id</td>
</tr>
<tr>
<td>e$C_allocation_list</td>
<td>error</td>
<td>return jop_job_allocation_list</td>
</tr>
<tr>
<td>e$C_job_class</td>
<td>error</td>
<td>return jop_job_class</td>
</tr>
</tbody>
</table>

6.3.3 The Process Object

The Process object appears at the third level of the UJPT hierarchy. Its primary function is to provide address space support and program image support for a set of execution threads, and to manage the set of process-level objects. The process object is the target of all accounting information. The process object can also act as a focal point for control operations.

There can be multiple processes in a job. Processes created as a result of job creation are top level processes. Once established, a process may cause the creation of other processes. These new processes are sub–processes, or child processes. Their creating processes are referred to as parent processes.

The Process object is implemented as a system level object in the "PROCESS$OBJECT_CONTAINER" object container.
6.3.3.1 Object Structure

Each process in the Mica system represents a set of execution threads and in some cases a set of sub-processes. The process object is responsible for managing the address spaces of its execution threads and for controlling the resource allocation limits of its execution threads.

The process object is split into a process object body and a process control block. The process object body contains the information necessary to maintain its position in the UJPT hierarchy, a task which includes coordinating its sub-process objects. The process control block contains the information necessary to manage the address space, to control the resource usage, and to pool accounting information of all of its execution threads. Example 6–13 illustrates the process object.

Example 6–13: Process Object Structure

```c

! Process Object Body

esProcess_object_body : RECORD
  p_obj_id : esObject_id;
p_job_pointer : POINTER esJob_object_body;
p_parent_process : POINTER esProcess_object_body;
p_process_flags : esProcess_flags;
p_process_queue : esLinked_list;
p_sub_process_queue : esLinked_list;
p_thread_queue_mutex : kmutex;
p_thread_count : integer;
p_thread_queue_hd : esLinked_list;
p_sub_process_queue_mutex : kmutex;
p_sub_process_count : integer;
p_sub_process_queue_hd : esLinked_list;
p_tcb : esProcess_control_block;
END RECORD;

! Object ID of process object
! Referenced pointer to owning job
! Referenced pointer to owning process, or NIL
! Process Flags
! List of jobs processes
! List of parents sub-processes
! Mutex for thread management
! Number of threads of the process
! List head of processes threads
! Mutex for sub-process management
! Number of sub-processes of the process
! List head of processes sub-processes
! Process Control Block

! Process Control Block

esProcess_control_block : RECORD
  pcb_usage_and_limits : esQuota_usage_and_limits;
  pcb_process_condir_id : esObject_id;

  pcb_process_alt_condir_id : esObject_id;

  pcb_process_public_object : esObject_id;
  pcb_process_private_object : esObject_id;
  pcb_accounting : esAccounting_summary;
  pcb_per_base : POINTER esProcess_control_region;
  pcb_process_control_pte : mmpte;
  pcb_tcb : POINTER mmpte;
  pcb_kernel_process_block : kprocess;
  pcb_exit_status_id : esObject_id;
  pcb_exit_status_pte : POINTER esExit_status_body;
  pcb_initial_thread_mutex : kmutex;
  pcb_process_allocation_list : esAllocation_list;

  ! Object Architecture Defined Container Directory Vector

  pcb_condir_mutex : ARRAY [esLevel_type] OF POINTER kdispatcher_object(mutex);
  pcb_condir_address : ARRAY [esLevel_type] OF POINTER esObject_header;
END RECORD;

! Current resources used/resource limits
! Process Level Container directory ID
! visible in an processes context
! Process Level Container directory ID
! visible in an arbitrary context
! Object ID of the process public container
! Object ID of the process private
! Process accounting summary
! User Readable Process Control Region
! Prototype PTE for seg 1 page table page
! Pointer to page table
! Kernel Process Block
! Exit Status Object ID for process
! Exit Status for Process
! mutex for initial threads kernel stack
! objects allocated to the process object

Example 6–13 Cont’d. on next page
Example 6–13 (Cont.): Process Object Structure

e$process_control_region : RECORD
    pcr_image_name : string(e$6c_max_image_name);
    pcr_total_number_of_threads : integer;
    pcr_number_running_threads : integer;
    pcr_object_id : e$object_id;
    pcr_protected_data_mutex : k$mutex;
    pcr_protected_data_hd : e$linked_list;
    pcr_data_block : POINTER anytype;
    pcr_data_block_length : integer;
    pcr_exit_handlers : e$exit_handler;
    pcr_execDispatch_table : e$dispatch_table;
END RECORD;

6.3.3.1.1 Resource Control

The process object maintains resource usage information for all of its threads. Unlike the job object, the process object's \textit{qual usage} values represent resources actively in use by its threads. Each time one of the process objects' threads consumes paged pool, the \textit{qual usage} field is incremented by the amount of pool actually used. This action is called \textit{pooling} the resource usage from the thread level to the process level.

During process object creation, the \textit{pcb_usage_and_limits.qual_limits} field of the process control block is set to the value of \textit{q_per_process_limits} from the user control block. The \textit{pcb_usage_and_limits.qual_usage} field of the process control block is then set to zero(), and the \textit{q_usage_and_limits.qual_usage} field of the job control block is incremented by \textit{q_per_process_limits} to reflect the resources allocated to the process. Once this resource shuffling operation has completed, the value of \textit{pcb_usage_and_limits.qual_limits} represents the amount of system resources available to the process object which can be consumed by all its thread objects.

While the above resource allocation scheme is the normal case, during process creation a parameter specifying the per-process limits for the process can be specified, altering the algorithm. This value simply overrides the value from \textit{q_per_process_limits} in the above example and applies to the newly created process.

6.3.3.1.2 Process Accounting

The process object maintains accounting information for all of its threads. Process accounting information is pooled from the thread level to the process level. Example 6–14 illustrates the types of information accounted for at the process level in the Mica system.

\textbf{NOTE}

Process accounting information is recorded with interlocked instructions, such that the information is always maintained in an up-to-date state.
Example 6-14: Process Accounting Structure

```plaintext
ACCOUNTING_SUMMARY : RECORD
  acct_total_page_faults : integer;
  acct_hard_page_faults : integer;
  acct_soft_page_faults : integer;
  acct_dzero_page_faults : integer;
  acct_com_page_faults : integer;
  acct_peak_virtual_memory : integer;
  acct_peak_working_set_size : integer;
  acct_start_time : estime_value;
  acct_end_time : estime_value;
  acct_page_file_usage : integer;
  acct_paged_pool_usage : integer;
  acct_nonpaged_pool_usage : integer;
  acct_cpu_and_io : escpu_and_io_summary;
END RECORD;

CPU and I/O accounting summary

An instance of this record exists in both the thread control block
and in the process control block. Updates to the poe version require interlocked
instructions. In the TCB version, only the execute Io counters will have to be updated
using interlocked instructions.

ESCPU_AND_IO_SUMMARY : RECORD
  cis_cpu_cycles : large_integer;
END RECORD;

6.3.3.2 Functional Interface

The Mica executive provides entry points capable of creating and deleting process objects, setting and
extracting various attributes of a Process object, and performing control operations on all threads
of a process. Control operations are Suspend/Resume Process, Hibernate/Wake Process, and Signal
Process.

As part of process-object creation, all of the necessary support data structures are created, including
a read only process control region (PCR), and a process-level object-container directory. The PCR is
part of the process's user-mode read-only address space. The Mica executive places information in
the PCR so that the process can read it without entering the system.
6.3.3.2.1 Process Creation

The `exec$create_process()` system service extends an existing UJPT hierarchy by causing the creation of a process object and a thread object. The newly created process object becomes a sub-process of the process above the calling thread. Example 6–15 illustrates the interface to `exec$create_process()`.

Example 6–15: Process Object Creation System Interface

```plaintext
PROCEDURE exec$create_process (  
OUT object_id : exec$object_id;  
IN object_parameters : exec$object_parameters = DEFAULT;  
IN process_record : exec$process_record;  
IN process_public_container : exec$object_id = DEFAULT;  
IN process_private_container : exec$object_id = DEFAULT;  
IN process_allocation_list : POINTER exec$allocation_list = NIL;  
IN process_data_block : POINTER quadword_data(*) CONFORM = NIL;  
IN thread_object_parameters : exec$object_parameters = DEFAULT;  
IN thread_record : exec$thread_record = DEFAULT;  
IN thread_allocation_list : POINTER exec$allocation_list = NIL;  
IN thread_data_block : POINTER quadword_data(*) CONFORM = NIL;  
IN thread_immediate_parameter1 : exec$thread_parameter = 0;  
IN thread_immediate_parameter2 : exec$thread_parameter = 0;  
IN thread_status : exec$object_id = DEFAULT;  
) RETURNS status;  
EXTERNAL;  
++
| Routine description:  
| Create a Process and thread object as specified by the parameters.  
| Arguments:  
| object_id Object ID of the resulting process object  
| object_parameters The object type independent parameters for the process object  
| process_record Attributes of the process being created  
| process_public_container Process level public container to be transferred into the process level container directory for the process. If not present then container comes up empty.  
| process_private_container Process level private container to be transferred into the process level container directory for the process. If not present then container comes up empty.  
| process_allocation_list Objects to be allocated to the process object. If not present then no objects are allocated to the process  
| process_data_block Arbitrary data block passed to the process  
| thread_object_parameters The object type independent parameters for the thread object  
| thread_record Attributes of the thread being created  
| thread_allocation_list Objects to be allocated to the thread object. If not present then no objects are allocated to the thread  
| thread_data_block Arbitrary data block passed to initial thread. Pointer in TCR, if pointer is NIL, then no data block was passed  
| thread_immediate_parameter1 Immediate parameter passed to thread through TCR  
| thread_immediate_parameter2 Immediate parameter passed to thread through TCR  
| thread_status Exit status object to be bound to the initial thread. If not present then the thread is created without an exit status object  
| process_status  
| Return value:  
| TBS  
|--

Process Structure 6–17
From the interface to `exec$create_process()`, it is clear that the `process_record` has an impact on the structure of the process being created. Example 6–16 illustrates the layout of the `process_record`.

**Example 6–16: Process Record Structure**

```plaintext
! The Process Record
!
exec$process_record : RECORD
    process_status_object : e$object_id;      ! Object ID of process's status object
    process_image_name : string(e$max_image_name);    ! Image name for process being created

    ! Per Process Resource limits. This value is used as the
    ! qual_limits value for the process object, and is deducted
    ! from the qual_usage field of the owning job object.
    ! A value of zero() in any one of fields means to use the
    ! corresponding value of the q_per_process_limit from the
    ! user structure.

    process_per_process_limits : e$quota_limits;    ! Resource limits for this process
END RECORD;
```

### 6.3.3.2.2 Process Deletion

The `exec$force_exit_process()` system service provides a mechanism for removing process objects from the system. The removal of a process has the following system-wide effects:

- All threads of the process are removed from the system.
- The amount of resources available to the process (`qual_limits–qual_usage`) is returned to the processes job object by decrementing `qual_usage` in the job object.
- If the process object is the last process owned by its job object, then the job object is removed from the system.

Example 6–17 illustrates the interface to `exec$force_exit_process()`.

**Example 6–17: Process Object Deletion System Interface**

```plaintext
PROCEDURE exec$force_exit_process (    
    IN process_object_id : exec$object_id = DEFAULT;
    IN exit_status : status;
) RETURNS status;
EXTERNAL;
```

**+**

**Routine description:**

Causes the Process object specified by `process_object_id` to be removed from the Ncsa system

**Arguments:**

- `process_object_id` the process object to be removed. If not specified, then the current process is assumed
- `exit_status` the reason that the process is force-exiting

**Return value:**

**+**

6–18 Process Structure
6.3.3.2.3 Get/Set Process Information

The exec$get_process_information() and exec$set_process_information() system services provide a mechanism to obtain and modify attributes of the specified process object. Example 6–18 illustrates the interfaces to the process object get/set system services.

Example 6–18: Get/Set Process Information System Interface

```c
PROCEDURE exec$get_process_information (  
    IN process_object_id : exec$object_id = DEFAULT;  
    IN process_get_items : POINTER exec$item_list_type;  
) RETURNS status;  
    EXTERNAL;  
++  
|  Routine description:  
|  Return information about the process object to the caller. The  
|  information returned is item list driven  
|  Arguments:  
|  process_object_id if present, the object ID of process object that is to be inspected  
|  otherwise, the process object of the calling thread is assumed  
|  process_get_items item list identifying process object information to be extracted  
|  Return value:  
|  TBS  
|  --
```

```c
PROCEDURE exec$set_process_information (  
    IN process_object_id : exec$object_id = DEFAULT;  
    IN process_get_items : POINTER exec$item_list_type;  
) RETURNS status;  
    EXTERNAL;  
++  
|  Routine description:  
|  Modify information in the process object. The  
|  information to be modified is item list driven  
|  Arguments:  
|  process_object_id if present, the object ID of process object that is to be modified  
|  otherwise, the process object of the calling thread is assumed  
|  process_get_items item list identifying process object information to be modified  
|  Return value:  
|  TBS  
|  --
```
Only certain pieces of the process object may be inspected or modified. Table 6-3 illustrates the possible item codes and the information read or written by using the item code.

<table>
<thead>
<tr>
<th>Item Code</th>
<th>Set Action</th>
<th>Get Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>e$C_job_id</td>
<td>error</td>
<td>return object ID of processes job object</td>
</tr>
<tr>
<td>e$C_parent_id</td>
<td>error</td>
<td>return object ID of processes parent process object</td>
</tr>
<tr>
<td>e$C_sub_process_count</td>
<td>error</td>
<td>return p_sub_process_count</td>
</tr>
<tr>
<td>e$C_sub_process_ids</td>
<td>error</td>
<td>return object ID's of sub_processes owned by process</td>
</tr>
<tr>
<td>e$C_thread_ids</td>
<td>error</td>
<td>return object ID's of threads owned by process</td>
</tr>
<tr>
<td>e$C_usage_and_limits</td>
<td>error</td>
<td>return pcb_usage_and_limits</td>
</tr>
<tr>
<td>e$C_process_limits</td>
<td>replace qual_limits</td>
<td>return qual_limits</td>
</tr>
<tr>
<td>e$C_process_condir_id</td>
<td>error</td>
<td>return pcb_process_condir_id</td>
</tr>
<tr>
<td>e$C_accounting</td>
<td>error</td>
<td>return pcb_accounting</td>
</tr>
<tr>
<td>e$C_pcr_base</td>
<td>error</td>
<td>return pcb_pcr_base</td>
</tr>
<tr>
<td>e$C_allocation_list</td>
<td>error</td>
<td>return pcb_process_allocation_list</td>
</tr>
<tr>
<td>e$C_protected_data</td>
<td>adds a block to pcr protected data list</td>
<td>error</td>
</tr>
</tbody>
</table>

6.3.3.2.4 Process Control Operations

Two process control operations exist in the Mica system to coordinate the execution of all threads of a process. The first provides a primitive which can alter the execution flow of another process by causing a condition to be raised in the target process. The second provides primitives to *block* and *unblock* the execution of the target process. In this latter technique, there are two classes of control operations. One class allows user-mode activity within the process to continue via user-mode AST routines, while the other class disables user-mode activity.

6.3.3.2.4.1 Process Signaling

The exec$signal_process() system service provides a mechanism to alter the execution flow of all threads of the process by causing a *condition* to be raised in the threads context.

**NOTE**

*Process signalling is implemented through user-mode ASTs; therefore, if ASTs are disabled then so are signals.*

Example 6-19 illustrates the interface to exec$signal_process().
Example 6–19: Signal Process System Interface

```plaintext
PROCEDURE exec$signal_process (  
    IN object_id : exec$object_id;  
    IN condition_value : exec$condition_value;  
    IN argument : longword CONFORM = DEFAULT;  
) RETURNS status;  
EXTERNAL;
```

```
++  
|  Routine description:  
|  Cause a condition of type condition_value to be raised in all 
|  threads owned by the process 
|  specified by object_id. The condition handler is passed 
|  argument.  
|  
| Arguments:  
|  
|  object_id  the object_id of the process to be signaled  
|  condition_value A descriptor for the condition to be raised in 
|  all threads  
|  argument of the target process  
|  
| Return value:  
|  
| TBS  
|--
```

6.3.3.2.4.2 Process Hibernate/Wake

The `exec$hibernate_process()` and `exec$wake_process()` provide a mechanism to block and unblock the execution flow of all threads within the target process. The block is implemented by causing all threads within the target process to issue a wait on the auto-clearing hibernate-event object within the thread control block. During the block, the only user-mode activity that is allowed is execution within user-mode AST routines; kernel-mode ASTs remain enabled. The unblock of the process is implemented by setting the auto-clearing hibernate event object within the thread control block of all threads of the target process. Example 6–20 illustrates the interfaces to `exec$hibernate_process()` and `exec$wake_process()`.

Example 6–20: Hibernate/Wake Process System Interface

```
| Hibernate Process  
| PROCEDURE exec$hibernate_process (  
|    IN object_id : exec$object_id;  
| ) RETURNS status;  
|EXTERNAL;
```

```
++  
| Routine description:  
| Cause all threads owned by the process specified by object_id to issue a wait on the 
| auto-clearing hibernate event object in their TCB. User mode AST's remain enabled  
| 
| Arguments:  
|  
|  object_id object ID of target process  
|  
| Return value:  
|  
| TBS  
|--
```

```
| Wake Process  
| PROCEDURE exec$wake_process (  
```

Example 6–20 Cont’d. on next page
Example 6–20 (Cont.): Hibernate/Wake Process System Interface

```
IN object_id : exec$object_id;
) RETURNS status;
EXTERNAL;

++
|
| Routine description:
|
| Cause all threads owned by the process specified by object_id to have their waits on the
| auto-clearing hibernate event object in their TCB to be satisfied by setting the event.
|
| Arguments:
| object_id object ID of target process
|
| Return value:
| TBS
|
|-
```

6.3.3.2.4.3 Process Suspend/Resume

The `exec$suspend_process()` and `exec$resume_process()` provide a mechanism to block and unblock the execution flow of all threads within the target process. The block is implemented by causing all threads within the target process to issue a wait on the auto-clearing suspend event object within the thread control block. During the block, no user-mode activity is possible; only kernel-mode normal and special AST routines may be executed. The unblock of the process is implemented by setting the auto-clearing suspend event object within the thread control block of all threads of the target process. Example 6–21 illustrates the interfaces to `exec$suspend_process()` and `exec$resume_process()`.

Example 6–21: Suspend/Resume Process System Interface

```
| Suspend Process
|
PROCEDURE exec$suspend_process (
    IN object_id : exec$object_id;
) RETURNS status;
EXTERNAL;

++
|
| Routine description:
|
| Cause all threads owned by the process specified by object_id to issue a wait on the
| auto-clearing suspend event object in their TCB. User mode AST's are disabled.
|
| Arguments:
| object_id object ID of target process
|
| Return value:
| TBS
|-
```

| Resume Process
|
PROCEDURE exec$resume_process (
    IN object_id : exec$object_id;
) RETURNS status;
EXTERNAL;

Example 6–21 Cont’d. on next page

6–22 Process Structure
Example 6-21 (Cont.): Suspend/Resume Process System Interface

#include

// Routine description:
// Cause all threads owned by the process specified by object_id to have their waits on the
// auto-clearing suspend event object in their TCB to be satisfied by setting the event.

// Arguments:
// object_id object ID of target process

// Return value:
// TBS

---

6.3.4 The Thread Object

The thread object appears at the lowest level of the UJPT hierarchy. Its primary function is to provide a thread of execution.

In addition, the thread object has the following functions:

- It is the schedulable entity in the Mica system.
- It maintains the processor state as it executes the program steps of an image.
- It is the consumer of resources. All accounting and resource limitation data structures reside in the thread’s process object, with the thread’s activity pooled to the process level. There is however a small set of accounting information that is stored on a per thread basis.
- It can act as a focal point for synchronization.

The thread object is implemented as a system level object in the "THREAD$OBJECT_CONTAINER" object container.

6.3.4.1 Object Structure

The thread object maintains the state of the processor as it moves through the program steps of the program image mapped into its processes address space.

The thread object is split into a thread object body and a thread control block. The thread object body contains information necessary to maintain the thread’s position within the UJPT hierarchy. The thread control block contains the information necessary to move the execution thread through the steps of the program image. Example 6-22 illustrates the thread object.
Example 6-22: Thread Object Structure

```
* Thread Object Body *

esThread : RECORD
   t_obj_id : eObject_id;
   t_process_pointer : POINTER esProcess_object_body;
   t_thread_flags : esThread_flags;
   t_thread_queue : esLinked_list;
   t_tcb : esThread_control_block;
END RECORD;

* Thread Control Block *

esThread_control_block : RECORD
   tcb_previous_mask : kProcessor_mode;
   tcb_kernel_thread_block : kThreadd;
   tcb_wait_timer : kTimer;
   tcb_hibernate_event : kEvent;
   tcb_suspend_event : kEvent;
   tcb_pcb_pointer : POINTER esProcess_control_block;
   tcb_active_thread_count_pointer : POINTER integer;
   tcb_tcb_base : POINTER esThread_control_region;
   tcb_exit_status_id : eObject_id;
   tcb_exit_status_ptr : POINTER esExit_status_body;
   tcb_exit_status_value : status;
   tcb_rundown_queue_mutex : kMutex;
   tcb_rundown_queue_head : esLinked_list;
   tcb_security_profile : esSecurity_profile;
   tcb_thread_allocation_list : esAllocation_list;
   tcb_thread_level_accounting : esCPU_and_IO_summary;

END RECORD;

* Memory Management Events *

tcb_initial_page_event : kEvent;
   tcb_secondary_page_event : kEvent;
   tcb_current_page_event : integer;

END RECORD;

* Thread Control Region *

The thread control region appears in the processes address space as user read only/ system read write

esThread_control_region : RECORD
   tcr_object_id : eObject_id;
   tcr_tcb Pointer : POINTER esThread_control_region;
   tcr_tcb_base : esProcess_control_block;
   tcr_initial_sp : esScalar_regisiter;
   tcr_stack_limit : esScalar_register;
   tcr_stack_base : esScalar_register;
   tcr_condition_initial_sp : esScalar_register;
   tcr_condition_stack_limit : esScalar_register;
   tcr_condition_stack_base : esScalar_register;
   tcr_exit_handlers : esExit_handlers;
   tcr_vectored_handlers : esVectored_handlers;

END RECORD;

* Initial Thread Parameters *

Example 6-22 Cont'd. on next page
```

6-24 Process Structure
Example 6–22 (Cont.): Thread Object Structure

```c
  tcr_block_data : POINTER anytype;
  tcr_block_data_length : integer;
  tcr_parameter1 : es$thread_parameter;
  tcr_parameter2 : es$thread_parameter;
END RECORD;
```

<table>
<thead>
<tr>
<th>Initial thread data block or NIL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Byte length of data block rounded to quadword</td>
</tr>
<tr>
<td>Immediate parameter / or zero()</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Immediate Parameter</th>
</tr>
</thead>
<tbody>
<tr>
<td>es$thread_parameter : es$scalar_register;</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Thread Entry Point</th>
</tr>
</thead>
<tbody>
<tr>
<td>es$thread_entry_point : PROCEDURE();</td>
</tr>
</tbody>
</table>

| Same size as a machine register |

### 6.3.4.2 Functional interface

The Mica executive provides entry points capable of creating, deleting, and controlling thread objects, in addition to setting and extracting various attributes of a thread object.

Thread object control services are Suspend/Resume thread, Hibernate/Wake thread, and Signal thread.

As part of Thread object creation, all of the necessary support data structures are created including the read-only thread control region (TCR), the read/write thread environment block (TEB), and user and kernel stacks. The TCR is part of the process's user-mode read-only address space. The Mica executive places information in the TCR so that the thread can read it without entering the system. The TEB is part of the user-mode thread architecture. The MICA executive initializes the TEB to point to the TCR.

#### 6.3.4.2.1 Thread Creation

The `exec$create_thread()` system service extends an existing UJPT hierarchy by causing the creation of a thread object. The newly created thread object begins execution within the address space of its process at a start address passed to the system interface. Example 6–23 illustrates the interface to `exec$create_thread()`.

Example 6–23: Thread Object Creation System Interface

```c
PROCEDURE exec$create_thread (  
  OUT object_id : exec$object_id;
  IN object_parameters : exec$object_parameters = DEFAULT;
  IN thread_procedure : exec$thread_entry_point;
  IN thread_record : exec$thread_record = DEFAULT;
  IN thread_allocation_list : POINTER exec$allocation_list = NIL;
  IN thread_data_block : POINTER quadword_data(*) CONFORM = NIL;
  IN thread_immediate_parameter1 : exec$thread_parameter = 0;
  IN thread_immediate_parameter2 : exec$thread_parameter = 0;
  IN thread_status : exec$object_id = DEFAULT;
)
```

Example 6–23 Cont’d. on next page
Example 6–23 (Cont.): Thread Object Creation System Interface

```plaintext
+RETURNS status;
EXTERNAL;
++
Routine description:
Create a thread object as specified by the parameters.
Arguments:
object_id Object ID of the resulting process object
object_parameters The object type independent parameters for the thread object
thread_record Attributes of the thread being created
thread_allocation_list Objects to be allocated to the thread object. If not present then
no objects are allocated to the thread
thread_data_block Arbitrary data block passed to initial thread. Pointer in TCR, if
pointer is NIL, then no data block was passed
thread_immediate_parameter1 Immediate parameter passed to thread through TCR
thread_immediate_parameter2 Immediate parameter passed to thread through TCR
thread_procedure pointer to thread entry point entry descriptor
thread_status Exit status object to be bound to the thread. If not present
then the thread is created without an exit status object

Return value:
+TBS
++
```

From the interface to exec$create_thread(), it is clear that the thread_record can have an impact on the structure of the thread being created. Example 6–24 illustrates the layout of the thread_record.

Example 6–24: Thread Record Structure

```plaintext
<table>
<thead>
<tr>
<th>The thread record</th>
</tr>
</thead>
<tbody>
<tr>
<td>e$thread_record : RECORD</td>
</tr>
<tr>
<td>thread_stack_size : integer;</td>
</tr>
<tr>
<td>thread_priority : k$combined_priority;</td>
</tr>
<tr>
<td>thread_affinity : k$affinity;</td>
</tr>
</tbody>
</table>
END RECORD; |
```

6.3.4.2.2 Thread Deletion

Thread deletion is the action which causes the removal of a thread object. The Mica system provides two mechanisms for deleting thread objects. The first mechanism, simple exit, will in some cases not cause the thread object to be removed; however, it is the normal path for thread exit when a thread wants to exit. The second mechanism, forced exit, will cause the thread object to be removed unconditionally. The forced exit path occurs when any thread wants the specified thread to exit.

The deletion of a thread object causes the thread's exit handlers to execute. In the simple exit case, exit handlers may run indefinitely, possibly never completing; thus, thread object may not occur. In the forced exit case, the thread's exit handlers are executed with a CPU time limit. If a time limit is exceeded, the next handler is executed. This technique guarantees that all exit handlers will be invoked and that afterwards thread object deletion will proceed. The exec$exit_thread() system interface provides the simple exit functionality. The exec$force_exit_thread() system service provides the forced-exit functionality.

When the last thread of a process is deleted, the process object is removed from the system.

Example 6–25 illustrates the interfaces to exec$exit_thread(), and exec$force_exit_thread().

6–26 Process Structure
Example 6–25: Thread Object Deletion System Interfaces

<table>
<thead>
<tr>
<th>Thread Exit System Service</th>
</tr>
</thead>
<tbody>
<tr>
<td>PROCEDURE exec$exit_thread (</td>
</tr>
<tr>
<td>IN exit_status : status;</td>
</tr>
<tr>
<td>);</td>
</tr>
<tr>
<td>++</td>
</tr>
<tr>
<td>Routine description:</td>
</tr>
<tr>
<td>Cause the deletion of the calling thread object. Place thread_status in the threads tcb at tcb_exit_status_value</td>
</tr>
<tr>
<td>Arguments:</td>
</tr>
<tr>
<td>thread_status the exit status of the thread</td>
</tr>
<tr>
<td>Return value:</td>
</tr>
<tr>
<td>none</td>
</tr>
<tr>
<td>--</td>
</tr>
<tr>
<td>Thread Force Exit System Service</td>
</tr>
<tr>
<td>PROCEDURE exec$force_exit_thread (</td>
</tr>
<tr>
<td>IN object_id : exec$object_id = DEFAULT;</td>
</tr>
<tr>
<td>IN exit_status : status;</td>
</tr>
<tr>
<td>) RETURNS status;</td>
</tr>
<tr>
<td>++</td>
</tr>
<tr>
<td>Routine description:</td>
</tr>
<tr>
<td>Cause the deletion of the thread object specified by object_id</td>
</tr>
<tr>
<td>Arguments:</td>
</tr>
<tr>
<td>object_id the object ID of the thread object being deleted. If not specified,</td>
</tr>
<tr>
<td>exit_status the reason that the thread is force-exiting</td>
</tr>
<tr>
<td>Return value:</td>
</tr>
<tr>
<td>TRS</td>
</tr>
<tr>
<td>--</td>
</tr>
</tbody>
</table>

6.3.4.2.3 Get/Set Thread Information

The exec$get_thread_information() and exec$set_thread_information() system services provide a mechanism to obtain and modify attributes of the specified thread object. Example 6–26 illustrates the interfaces to the thread object get/set system services.
Example 6-26: Get/Set Thread Information System Interface

```c
PROCEDURE execGet_thread_information {
    IN thread_object_id : execobject_id = DEFAULT;
    IN thread_get_items : POINTER execItem_list_type;
} RETURNS status;
EXTERNAL;

+++ Routine description:
| Return information about the thread object to the caller. The
| information returned is item list driven

Arguments:
| thread_object_id if present, the object id of thread object that is to be inspected
| thread_get_items otherwise, the calling thread is assumed

Return value:
| TBS

---
```

```c
PROCEDURE execSet_thread_information {
    IN thread_object_id : execobject_id = DEFAULT;
    IN thread_get_items : POINTER execItem_list_type;
} RETURNS status;
EXTERNAL;

+++ Routine description:
| Modify information in the thread object. The
| information to be modified is item list driven

Arguments:
| thread_object_id if present, the object ID of thread object that is to be modified
| thread_get_items otherwise, the calling thread is assumed

Return value:
| TBS

---
```

Only certain pieces of the thread object may be inspected or modified. Table 6-4 illustrates the possible item codes and the information read or written by using the item code.

<table>
<thead>
<tr>
<th>Table 6-4: Get/Set Thread Information Item Codes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Item Code</td>
</tr>
<tr>
<td>-----------</td>
</tr>
<tr>
<td>e$C_process_id</td>
</tr>
<tr>
<td>e$C_tcr_base</td>
</tr>
<tr>
<td>e$C_tcr_start_address</td>
</tr>
<tr>
<td>e$C_allocation_list</td>
</tr>
</tbody>
</table>

6-28 Process Structure
6.3.4.2.4 Thread Control Operations

Two thread control operations exist in the Mica system to coordinate the execution of threads. The first provides a primitive which can alter the execution flow of another thread by causing a condition to be raised in the target thread. The second provides primitives to block and unblock the execution of the target thread. In this latter technique, there are two classes of control operations. One class allows user-mode activity within the thread to continue via user-mode AST routines, while the other class disables user-mode activity.

6.3.4.2.4.1 Thread Signaling

The exec$signal_thread() system service provides a mechanism to alter the execution flow of a thread by causing a condition to be raised in the context of the target thread.

NOTE

The thread signalling mechanism is implemented through user-mode ASTs; therefore, if ASTs are disabled, then so are signals.

Example 6–27 illustrates the interface to exec$signal_thread().

Example 6–27: Signal Thread System Interface

PROCEDURE exec$signal_thread {
    IN object_id : exec$object_id;
    IN condition_value : exec$condition_value;
    IN argument : longword CONFORM = DEFAULT;
    ) RETURNS status;
    EXTERNAL;
}

|++
| | Routine description:
| | Cause a condition of type condition_value to be raised in the thread
| | specified by object_id. The condition handler is passed argument.
| | Arguments:
| | object_id the object_id of the thread to be signaled
| | condition_value A descriptor for the condition to be raised in the target thread
| | argument If present, the value that is passed to the condition handler
| | Return value:
| | TBS
| | --

6.3.4.2.4.2 Thread Hibernate/Wake

The exec$hibernate_thread() and exec$wake_thread() provide a mechanism to block and unblock the execution flow of a thread. The block is implemented by causing the thread to issue a wait on the auto-clearing hibernate event object within the thread control block. During the block, the only user-mode activity that is allowed is execution within user-mode AST routines. The unblock of the thread is implemented by setting the auto-clearing hibernate event object within the thread control block. Example 6–28 illustrates the interfaces to exec$hibernate_thread() and exec$wake_thread().
Example 6-28: Hibernate/Wake Thread System Interface

**Hibernate Thread**

PROCEDURE exec$hibernate_thread (  
    IN object_id : exec$object_id;  
) RETURNS status;  
EXTERNAL;

**++**

Routine description:

Cause the thread specified by object_id to issue a wait on the auto-clearing hibernate event object in the TCB. User mode AST's remain enabled.

Arguments:

- object_id: object ID of target thread

Return value:

- TBS

**--**

**Wake Thread**

PROCEDURE exec$wake_thread (  
    IN object_id : exec$object_id;  
) RETURNS status;  
EXTERNAL;

**++**

Routine description:

Cause the thread specified by object_id to have the wait on the auto-clearing hibernate event object in the TCB to be satisfied by setting the event.

Arguments:

- object_id: object ID of target thread

Return value:

- TBS

**--**

6.3.4.2.4.3 Thread Suspend/Resume

The exec$suspend_thread() and exec$resume_thread() provide a mechanism to block and unblock the execution flow of the target thread. The block is implemented by causing the thread to issue a wait on the auto-clearing suspend-event object within the thread control block. During the block, no user-mode activity is possible. Only kernel-mode normal and special AST routines may be executed. The unblock of the thread is implemented by setting the auto-clearing suspend-event object within the thread control block. Example 6-29 illustrates the interfaces to exec$suspend_thread() and exec$resume_thread().

6-30  Process Structure
Example 6-29: Suspend/Resume Thread System Interface

PROCEDURE exec$suspend_thread ()
  IN object_id : exec$object_id;
  ) RETURNS status;
  EXTERNAL;

+++ Routine description:
  Cause the thread specified by object_id to issue a wait on the
  auto-clearing suspend event object in the TCB. User mode AST's are disabled.
  Arguments:
  object_id    object ID of target thread
  Return value:
  TBS
  --

RESUME THREAD

PROCEDURE exec$resume_thread ()
  IN object_id : exec$object_id;
  ) RETURNS status;
  EXTERNAL;

+++ Routine description:
  Cause the thread specified by object_id to have the wait on the
  auto-clearing suspend event object in the TCB to be satisfied by setting the event.
  Arguments:
  object_id    object ID of target thread
  Return value:
  TBS
  --

6.3.4.2.4.4 Hibernate and Suspend Comparison

Both the exec$hibernate_thread() system service, and the exec$suspend_thread() system service block the execution of the specified thread. The difference between these two types of blocked states is the ability of the blocked thread to receive and execute in the context of user-mode ASTs. Threads that are blocked due to the exec$hibernate_thread() system service are able to receive and execute in the context of user-mode ASTs; threads that are blocked due to the exec$suspend_thread() system service are not.

6.4 UJPT Object Linkages

The UJPT hierarchy is bound together through the existence of object IDs and referenced pointers. The following section describes the implementation of the object linkages, the steps of hierarchy creation, and the actions which lead to the collapse of a UJPT hierarchy. This section does not describe process or thread creation in terms of address space creation or the intricate details of kernel, memory management, or object architecture interactions.
6.4.1 Linkage Structure

The UJPT object linkage structure requires that objects in lower levels of the hierarchy point to the object immediately above them using a referenced pointer. The reference pointer guarantees the existence of the higher-level object for the life of the lower-level object. Figure 6-1 illustrates a complex UJPT hierarchy consisting of a user object, a job object, and a process object consisting of two immediate threads and a sub-process object with a single thread.

**Figure 6-1: Complex UJPT Hierarchy**

```
User.0  2,1  ID0
     Job C1
      2,1  ID1
       Process C1
        Process 0  4,1  ID2
                          Sub Process C1
                          1  Thread C1
                             2,1  ID3
                             Thread 0
                              2,1  ID4
                              Thread 1
                              2,1  ID5
                              Process 1
                              2,1  ID6
                              Thread 2
```

X,Y = Pointer Count, Object ID Count

= Referenced Pointer

6.4.2 Hierarchy Creation

The creation of a UJPT hierarchy is triggered by the `exec$create_user()` system service. At this time, a hierarchy is either created or extended, depending on the existence of a user object representing the Mica user specified in the `user_record.user_username` field of the `user_record` parameter.
The following steps occur during the creation of a UJPT hierarchy.

1. Determine if a user object exists for user_record.user_username. If the object exists, then obtain a referenced pointer to the user object. Otherwise, create the user object in the system container directory, initialize the user object with the information from the user_record and then obtain a referenced pointer to the user object.

2. Create the job object in the system container directory and obtain a referenced pointer to the job object. Initialize the job object according to the following tasks.
   - Set j_obj_id equal to the object ID of the job being created.
   - Set j_user_pointer to the referenced pointer of the proper user object.
   - Link the job object to the user object's u_job_queue_hd, and initialize the j_process... fields of the job object.
   - Create the job-level container directory, and populate it with the job_initial_container parameter.

3. Create the process object in the system container directory and obtain a referenced pointer to the process object. Initialize the process object according to the following tasks.
   - Set p_obj_id equal to the object ID of the process being created.
   - Set p_job_pointer to the referenced pointer of the proper job object.
   - Link the process object to the job object's j_process_queue_hd.
   - Initialize the p_thread... fields and p_sub_process... fields of the job object.
   - Create the process level container directory, and populate it with the process_public_container parameter and the process_private_container parameter.

4. Create the thread object in the system container directory.

5. Obtain a referenced pointer to the thread object.

6. Initialize the thread object such that t_obj_id contains the object ID of the thread, and t_process_pointer contains the referenced pointer to the proper process object.

7. Link the thread object to the process object's p_thread_queue_hd.

6.4.3 Hierarchy Collapse/Deletion

The collapse of a UJPT hierarchy can be triggered by force-exiting any component of a hierarchy. The ultimate collapse is always the result of a thread's exit, whether it be a forced exit or a voluntary exit.

The forced exit of a component in the UJPT hierarchy eventually causes all threads beneath that object to exit. The following actions occur during a thread exit.

- If the exiting thread is the last thread in its process, then cause the process to exit by removing its object ID.
- If the exiting process has any sub-processes, then cause its sub-processes to exit by force-exiting them.
- If the exiting process is the last process in its job, then cause the job to exit by removing its object ID.
- If the exiting job is the last job in its user, then cause the user to exit by removing its object ID.
6.4.3.1 Force-Exit Routines

Each component of the hierarchy provides a force-exit interface as part of its primitive object service routines. The basic action performed in these routines is the forced exit of the object's sub-objects.

6.4.3.1.1 User-Object Force-Exit Routine

The user-object force-exit routine is responsible for causing the forced exit of all of its job objects. This is implemented by setting its force-exit in-progress flag, and looping over the linked list of its job objects headed by u_job_queue_hd and a force-exit of that job via $force_exit_job().

6.4.3.1.2 Job-Object Force-Exit Routine

The job object force-exit routine is responsible for causing the forced exit of all of its process objects. This is implemented by setting its force-exit in progress flag, looping over the linked list of its process objects headed by j_process_queue_hd, and causing a forced exit of that process via $force_exit_process().

6.4.3.1.3 Process-Object Force-Exit Routine

The process object force-exit routine is responsible for causing the removal of all of its thread objects and sub-processes represented as process objects. This is implemented by setting its force-exit in progress flag, and looping over the linked list of its thread objects headed by p_thread_queue_hd and causing a force-exit of that thread via $force_exit_thread(). Then the routine loops over the linked list of sub-processes headed by p_sub_process_queue_hd and causes a forced exit of that process via $force_exit_process().

6.4.3.1.4 Thread Object Force Exit Routine

The routine occurs in two phases. The first phase is to cleanly enter the exiting thread's context to begin the thread exit. The second phase is to complete the exit of the thread by calling exec$exit_thread(), an action which starts the second phase of hierarchy collapse and finally brings the "exiting" thread out of the system. Before starting the forced-exit processing, the force-exit in-progress flag is set in the thread object.

During a thread forced-exit, there is a moment when control is returned to the original caller of exec$exit_thread() even though the thread to be exited is still part of the system. The exit is considered complete with respect to the caller after the system has delivered an AST to the exiting thread that will cause the thread itself to exit. The exit is complete with respect to the exiting thread once the thread has issued its call to k$terminate_thread();

6.4.3.1.4.1 Thread Context Entry

To force-exit a thread, that thread's context must be entered in a controlled manner in a "trusted" user-mode routine. This is achieved by delivering a user-mode AST to the thread. The target procedure of the AST is a routine that is part of the Mica executive but is executed in user mode. The AST target procedure is the function $in_context_force_exit(). The purpose of this function is to bring the thread into a "clean" state so that it can complete its exit. The following steps occur in $in_context_force_exit():

- The thread issues an $unwind() specifying an exit unwind.
- Once the unwind has completed, the thread issues a call to exec$exit_thread().

6–34 Process Structure
6.4.3.1.4.2 Thread Exit

The second phase of thread exit processing begins at the entry point exec$exit_thread(). The purpose of this function is to execute all of the exit handlers for the thread and, when completed, to bring the thread object out of the system. The following steps occur in exec$exit_thread():

- Dequeue the first exit handler from the thread control region.
- If the thread is in the force-exit in progress state, then establish a CPU time quota for the thread.

**NOTE**

The thread-exit CPU time quota is on accumulated user-mode CPU time. It is not an elapsed time limit.

- If the CPU time quota expires, then deliver a user-mode AST to the thread. The target procedure of the AST is executive code that runs in "trusted" user-mode at the e$exit_handler_quota_expire(). This entry point causes the termination of the current exit handler and begins the next by calling exec$exit_thread().
- Vector to the exit handler in user-mode.
- If no more exit handlers for the thread exist, then remove the object ID of the thread by calling e$remove_object_id(), passing it the object ID of the thread stored in t_obj_id in the thread object body. This action begins the second phase of hierarchy collapse by causing the execution of the affected object's remove routines. If there are more exit handlers, then repeat the above steps.
- After completion of e$remove_object_id(), the thread removes itself from the system by calling k$terminate_thread(). This action begins the third phase of hierarchy collapse by causing the execution of the affected object's delete routines.

6.4.3.2 Object Remove Routines

The object remove routines are called when the objhdr$object_id_count within the object header decrements to zero. This occurs during the second phase of hierarchy collapse as a result of a call to e$remove_object_id() for the "exiting" object. Object remove routines are always executed in the context of the object being removed.

**NOTE**

In order to ensure the above context restrictions, objects within the UJPT hierarchy may not have alias object IDs, and their ACLs are such that only the function exec$exit_thread() is capable of removing their object IDs.

Assuming the UJPT hierarchy from Figure 6-1, the following legal contexts exist to execute the remove routines for the hierarchy.

- Thread.0 will execute its remove routine in the context of thread.0.
- Thread.1 will execute its remove routine in the context of thread.1.
- Thread.2 will execute its remove routine in the context of thread.2.
- Process.0 could execute its remove routine in either the context of thread.0, or thread.1. The context would be determined by the context of the last thread to begin the second phase of exit.
- Process.1 will execute its remove routine in the context of thread.2.
- Job.0 will execute its remove routine in the context that was used to execute process.0's remove routine.
- User.0 will execute its remove routine in the context that was used to execute job.0's remove routine.
6.4.3.2.1 User-Object Remove Routine

The user-object remove routine performs no actions related to hierarchy collapse.

6.4.3.2.2 Job Object Remove Routine

The job-object remove routine is responsible for breaking the link between itself and its user object. If the job object is the last object of its user object then it must guarantee the removal of the user object. This occurs as follows:

- The job object is de-linked from the u_job_queue_hd in the user object pointed to by j_user_pointer.
- If the u_job_count field is decremented to zero by this action, then the user object is removed by calling e$remove_object_id() specifying the object ID of the user object (u_obj_id) stored in the user object body.

6.4.3.2.3 Process Object Remove Routine

The process object remove routine is responsible for breaking the link between itself and its job object, and if the process is a sub-process, it must break the link between itself and its parent process i.e. the process above it. Two different paths are followed during the process remove routine. The following occurs in the remove routine for a process without a parent.

- The process object is de-linked from the j_process_queue_hd in the job object pointed to by p_job_pointer.
- If the j_process_count field is decremented to zero by this action, then the job object is removed by calling e$remove_object_id() specifying the object ID of the job object (j_obj_id) stored in the job object body.

The remove routine for a sub-process i.e. a process with a parent simply de-links itself from the p_sub_process_queue_hd in the process object pointed to by p_parent_pointer.

6.4.3.2.4 Thread Object Remove Routine

The thread object remove routine is responsible for breaking the link between itself, and its process object. If the thread object is the last object of its process object then it must guarantee the removal of the process object. This occurs as follows:

- The thread object is de-linked from the p_thread_queue_hd in the process object pointed to by t_process_pointer.
- If the p_thread_count field is decremented to zero by this action, then the process object is removed by calling e$remove_object_id() specifying the object ID of the process object (p_obj_id) stored in the process object body.

6.4.3.3 Object Delete Routines

The object delete routines are called as a result of the objhdr$pointer_count field decrementing to zero. This occurs during the third phase of hierarchy collapse as a result of the call to k$terminate_thread() in exec$exit_thread().

The function of k$terminate_thread() is to remove the thread from the system. This is accomplished by queuing a pointer to the thread object to a queue served by a system thread running e$terminate_thread(). This thread is responsible for dereferencing the thread object which begins the third phase of hierarchy collapse.
Object delete routines always execute in the context of the system thread running $\texttt{\texttt{\$terminate\_thread}}()$.

**NOTE**

At the time that $\texttt{\texttt{\$terminate\_thread}}()$ is called, the thread object's $\texttt{\texttt{\$objhdr\$pointer\_count}}$ is 1, and the $\texttt{\texttt{\$objhdr\$object\_id\_count}}$ is 0.

6.4.3.3.1 User-Object Delete Routine

The user-object delete routine performs no actions related to hierarchy collapse.

6.4.3.3.2 Job-Object Delete Routine

The job-object delete routine simply dereferences its user object by calling $\texttt{\texttt{\$dereference\_object}}()$ passing it the referenced pointer to the user object stored in $\texttt{j\_user\_pointer}$.

6.4.3.3.3 Process-Object Delete Routine

The process-object delete routine performs the following actions:

- If the process has a parent process, its parent process object is dereferenced by calling $\texttt{\texttt{\$dereference\_object}}()$, passing it the referenced pointer to the parent process object stored in $\texttt{p\_parent\_pointer}$.
- The job object is dereferenced by calling $\texttt{\texttt{\$dereference\_object}}()$, passing it the referenced pointer to the job object stored in $\texttt{p\_job\_pointer}$.

6.4.3.3.4 Thread-Object Delete Routine

The thread-object delete routine simply dereferences its process object by calling $\texttt{\texttt{\$dereference\_object}}()$, passing it the referenced pointer to the process object stored in $\texttt{t\_process\_pointer}$.

6.5 Address Space and Execution Threads

Execution threads exist within a context which includes an address space and processor state. The creation and deletion of execution threads involves heavy interactions with the Mica kernel and memory management subsystems. This section describes execution thread creation and deletion in terms of its interactions with the Mica kernel, executive, and memory management subsystems. Interactions with the object architecture are not discussed.

6.5.1 Creation

The creation of an execution thread has two distinct paths.

The first path occurs when an execution thread is being created, an action which requires the creation of both an address space and a processor state. This path is a result of an $\texttt{\texttt{\$exec\_create\_user}}()$, an $\texttt{\texttt{\$exec\_create\_job}}()$, or $\texttt{\texttt{\$exec\_create\_process}}()$ system service. This path is known as initial thread creation.

The second path occurs when an execution thread is being created within an existing address space. The only context that needs to be established is the processor state. This path occurs as a result of an $\texttt{\texttt{\$exec\_create\_thread}}()$ system service and is known as subsequent thread creation.
6.5.1.1 Initial Thread Creation

During initial thread creation, the following actions occur.

- An address space must be created and initialized.
- A transition to the new thread's partial context must occur.
- Both thread- and process-control region address space must be created and initialized.
- The program image for the new process must be mapped into the process address space.
- The thread must begin execution at the program image starting address

6.5.1.1.1 Address Space Creation

The creation of a Mica address space occurs as a result of a call to $create_process_address_space()$. Example 6-30 illustrates the interface to this function.

Example 6-30: Address Space Creation

```c
PROCEDURE $create_process_address_space ( 
  IN process_control_pte : POINTER mmppte; 
  OUT ptbr : integer; 
  OUT new_kernel_stack_pointer : POINTER anytype CONFORM; 
  OUT valid_in_this_process_pointer : POINTER anytype CONFORM;
); 
EXTERNAL;
```

++

Routine description:

This routine creates the foundation of a process address space. Pages are allocated for the segment 1 page table, the segment 2 page table for the control region, the kernel stack and the working set list.

NO ADDRESSES WITHIN THE ADDRESS SPACE ARE VALID, THIS INCLUDES THE KERNEL STACK POINTER WHICH IS RETURNED.

Once an address space foundation has been created, $initialize_thread$ is invoked to create the initial thread running within this new address space.

Arguments:

- IN process_control_pte - pointer to the process_control_pte in the process control block. Upon return the prototype PTE referred to by process_control_pte will contain the prototype PTE for the segment 1 page table page. The PPN database PTP element will contain this address (process_control_pte) so it must be in non-paged system space.

- OUT ptbr - the value to be used for the page table base register

- OUT kernel_stack_pointer - the value to be used for the kernel stack pointer

- OUT valid_in_this_process_pointer - the value to be used for the kernel stack pointer while in the calling context

Return value:

- none.

The created address space is only valid in the context of the new thread. The next phase of address space creation occurs in the context of the new thread.
6.5.1.1.2 Execution Thread Creation

Once the address space for the initial thread is created, the thread must be started in its context. This occurs by calls to the kernel interface $initialize_thread(). After the completion of $initialize_thread(), the new thread is eligible to run in its own context, and the calling thread considers the thread creation complete. The calling thread then calls $release_temp_kernel_stack() freeing the valid_in_this_process_pointer temporary kernel stack.

The new thread begins execution at $initialize_thread_startup(). Example 6–31 illustrates the entry point for all initial threads.

Example 6–31: Initial Thread Entry Point

PROCEDURE $initialize_thread_startup();
EXTERNAL;
++
|| Routine description:
| The entry point for all initial threads. This routine is responsible for completing an
| execution thread which involves
| o initializing the thread's address space
| o creating and initializing the control region memory pool
| o initializing the pcr and tcr
| o mapping the program image into the new address space
| o starting the thread at the image entry point
| Arguments:
| none
| Return value:
| none
|--

6.5.1.1.2.1 Address Space Initialization

The first action performed by $initialize_thread_startup() is the initialization of the process address space. This action makes it possible for the thread to begin taking page faults within its address space. Address space initialization is accomplished by calling $initialize_address_space(). Example 6–32 illustrates the interface to $initialize_address_space().

Example 6–32: Address Space Initialization

PROCEDURE $initialize_address_space ()
IN working_set_extent : integer;
IN working_set_quota : integer;
);
EXTERNAL;
++
|| Routine description:
| This routine initializes an address space which was previously
| created by $create_process_address_space.
| It must now be running in the non paged portion of the exec
| with the newly created address space mapped. No page faults
| may be taken until this routine has been invoked.
| This routine will create the working set list, mark the control
| region, kernel stack, and working set list as locked in the
| working set.
| The arguments are derived from the process control block
| qnl_working_set_limit and qnl_working_set_extent fields of
| pcb_usage_and_limits structure.
| Arguments:

Example 6–32 Cont'd. on next page
Example 6-32 (Cont.): Address Space Initialization

| IN working_set_limit - maximum size of the working set. |
| IN working_set_quota - current size of the working set. |
| Return value: |
| none - it had better work. |

6.5.1.2.2 Control Region Initialization

Once the process address space has been initialized, the control region memory pool becomes valid. The control region is at a fixed virtual address within the process's address space and is user read-only, kernel read/write.

Once the control region pool has been created, a process control region and thread control region are allocated from the control region pool. The control regions are then initialized by copying dummy control regions allocated from non-paged pool to the real control regions. Finally, the thread control region is linked to its thread control block, and the process control region is linked to the process control block and the thread control region.

6.5.1.2.3 Program Image Mapping

The program image to be executed must be mapped into the newly created process address space. This occurs by transitioning into user-mode at the entry point exec$program_image_startup().

The function of exec$program_image_startup() is to map the program image and cause it to begin execution at the image start address. To map the image, the function exec$map_image() is called passing it the image name stored in its process control region. Once mapped, the thread startup address stored in the thread control region is set using exec$set_thread_information(). The image is then called. The initial thread parameters may be found in the thread control region.

6.5.1.2 Subsequent Thread Creation

During subsequent thread creation the following must occur:

- Creation of a kernel mode and user mode stack for the thread.
- Creation and initialization of the thread control region.
- Transition to the new thread's context at the proper start address.

6.5.1.2.1 Thread Stack Creation

The creation of a kernel and user mode stack for the new thread occurs as a result of calling exec$create_thread_stacks().

6.5.1.2.2 Control Region Initialization

A thread control region is allocated for the new thread from the control region pool of the calling thread's process. The thread control region is then initialized with the values obtained from the exec$create_thread() parameters. The thread control region is then linked to the thread control block and is set to point to the proper process control region.
6.5.1.2.3 Transition to new Thread

The final steps in subsequent thread creation require that the thread be started in its context. This is achieved by making a call to $initialize_thread$. After the completion of the call to $initialize_thread$, the new thread is eligible to be run in its own context, and the calling thread assumes that the thread creation has completed.

The new thread begins execution at $subsequent_thread_startup$. This entry point simply forces a transition to user-mode at the address specified by the thread control blocks $tcr_start_address_field$.

6.5.2 Deletion

Address space and execution thread deletion happen as part of the process object and thread object delete routines.

6.5.2.1 Execution Thread Deletion

Execution thread deletion happens in two phases. The first phase is executed within the context of the terminating thread and is responsible for thread resource cleanup. The second phase occurs outside the context of the calling thread and is responsible for the deletion of the kernel stack of the terminating thread.

NOTE

The context restrictions are enforced by the lack of alias object IDs on components of the UOJT hierarchy, and through restrictions on the removal of objects within the hierarchy.

6.5.2.1.1 In-Context Thread Deletion

In-context thread deletion involves returning to the system all resources owned by the thread. This may include AST control blocks, IO request packets, and other outstanding system resources. All mutexes owned by the thread must be dealt with, and the thread control region must be returned to the control region pool of its process. These actions occur as part of the thread object's remove routine.

The second phase of execution thread deletion is then started by calling the kernel primitive $terminate_thread$.

6.5.2.1.2 Out of Context Thread Deletion

The call to $terminate_thread$ is responsible for queuing a terminate-thread descriptor on a queue served by the system thread responsible for out-of-context thread deletion. The server causes the thread object’s delete routine to be executed by dereferencing the pointer to the thread object.

The thread object delete routine deletes the kernel stack of the terminating thread by calling $delete_thread_stack$.

At the end of out-of-context thread deletion, all data structures that represent the thread are returned to the system. This includes the entire thread object and thread control block.

NOTE

The thread control region is deallocated during in context thread deletion because it must refer to the thread's process address space.
6.5.2.2 Address Space Deletion

If the terminating execution thread is the last thread of its process, then the address space of the process must also be deleted. This occurs in the process delete routine.

NOTE

Address space, as used above, means address-space management data structures such as page tables, working set lists, and the last thread's kernel stack.

The user-mode address space is deleted mostly as a result of removing the process level container directory, since user-mode address space is represented as section objects.

The process delete routine calls e$delete_process_address_space(), specifying the page table base register value from process object body.

6.6 Exit Status

The exit status mechanism in the Mica system supports the ability to obtain the exit status from a process and, in some cases, from an individual thread within a process.

The exit status mechanism is coordinated through the exit status object.

6.6.1 Object Structure

The exit status object contains information describing the termination state of the object it is bound to. Example 6–33 illustrates the layout of the exit status object.

Example 6–33: Exit Status Object Structure

```
! Exit Status Object Body
!
e$exit_status_body : RECORD
    e$exit_status_summary : e$exit_status_summary;
    e$exit_status_event : k$event;
END RECORD;
!
! Exit Status Summary
!
e$exit_status_summary : RECORD
    status_bound_object_type : e$status_object_types;
    status_bound_object_id : e$object_id;
    status_value : status;
    status_string_pointer : POINTER string(*);
END RECORD;
```

6.6.2 Functional Interface

The Mica executive provides interfaces to create and obtain information from exit status objects.
6.6.2.1 Exit Status Object Creation

Exit status objects are created by the exec$create_exit_status() system service. Exit status objects are created with an initial status_value of status$pending, and an initial status_string_pointer of NIL. and are not bound to either a process or a thread object. The object binding occurs during thread and process object creation. Example 6–34 illustrates the interface to exec$create_exit_status().

Example 6–34: Exit Status Object Creation System Interface

```c
PROCEDURE exec$create_exit_status (
    OUT object_id : exec$object_id;
    IN object_parameters : exec$object_parameters = DEFAULT;
) RETURNS status;
EXTERNAL;
++

| Routine description:
| Create an exit status object
| The default container for exit status objects is the process private container
| Arguments:
| object_id The object ID of the created exit status object
| object_parameters The object type independent parameters for the exit status object
| Return value:
| TBS
|--
```

6.6.2.2 Get/Set Exit Status Information

The exec$get_exit_status_info and exec$set_exit_status_info system services provide a mechanism to obtain and to modify attributes of the specified exit status object. Example 6–35 illustrates the interfaces to the exit status object get/set system services.

Example 6–35: Get/Set Exit Status Information System Interface

```c
PROCEDURE exec$get_exit_status_info (
    IN object_id : exec$object_id = DEFAULT;
    IN process_status_object : boolean = true;
    IN items : POINTER exec$item_list_type;
) RETURNS status;
EXTERNAL;
++

| Routine description:
| Return information about the exit status object to the caller. The information returned is item list driven
| Arguments:
| object_id - the object id of the exit status object to get information from
| process_status_object - if true then use the process objects exit status object, otherwise use the thread objects exit status object. (Ignored if object ID is valid).
| items - item list identifying exit status object information to be extracted
| Return value:
| TBS
|--
```

Example 6–35 Cont’d. on next page
Example 6-35 (Cont.):  Get/Set Exit Status Information System Interface

```
PROCEDURE exec$set_exit_status_info (  
    IN process_status_object : boolean = true;  
    IN items : POINTER exec$item_list_type;  
) RETURNS status;  
EXTERNAL;
```

Only certain pieces of the exit status object may be inspected or modified. Table 6-5 illustrates the possible item codes and the information read or written when using the item code.

<table>
<thead>
<tr>
<th>Item Code</th>
<th>Set Action</th>
<th>Get Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>e$c_status_value</td>
<td>error</td>
<td>returns the current status_value</td>
</tr>
<tr>
<td>e$c_status_string</td>
<td>sets the status string to the specified string</td>
<td>returns the current value of the status string</td>
</tr>
<tr>
<td>e$c_status_string_set</td>
<td>error</td>
<td>returns the value of the status_string_pointer field</td>
</tr>
<tr>
<td>e$c_status_summary</td>
<td>error</td>
<td>returns the current exit status summary</td>
</tr>
</tbody>
</table>

6.6.3 Usage

Exit status objects are used to report the exit status of exiting processes and exiting threads. Each thread in the Mica system may optionally be bound to an exit status object. The binding occurs during the creation of the thread. Each process in the Mica system is bound to an exit status object. The binding occurs during the creation of the process. Exit status objects are waitable objects. They are initialized in an un-signaled state, and remain in that state until the object that they are bound to is removed from the system.
6.6.3.1 Thread Exit Status Object Usage

If the `thread_status` parameter is specified during the direct or indirect creation of a thread, then the thread is bound to the specified exit status object. The exit status object is made signaled during the object remove routine for an exiting thread. This occurs as follows:

- Set the `status_value` field in the exit status object to the value stored in the thread control block `tcb_exit_status_value` field.

**NOTE**

This step actually happens prior to the threads object remove routine, once for each call to `exec$exit_thread()`.

- Set the `status_value` field in the exit status object of the threads' process to the value stored in the thread control block `tcb_exit_status_value` field.

**NOTE**

This step actually happens prior to the threads object remove routine, once for each call to `exec$exit_thread()`.

- Signal the `es_exit_status_event` in the exit status object bound to the exiting thread.

If the exiting thread is not bound to an exit status object, then the following occurs.

- Set the `status_value` field in the exit status object of the threads' process to the value stored in the thread control block `tcb_exit_status_value` field.

**NOTE**

This step actually happens prior to the threads object remove routine, once for each call to `exec$exit_thread()`.

6.6.3.2 Process Exit Status Object Usage

Each process in the Mica system is bound to an exit status object. During the object remove routine for a process object, the process exit status object is signaled by setting the `es_exit_status_event` in the exit status object bound to the exiting process. The `status_value` field were previously set during the individual thread exits for all of the processes.

**NOTE**

It is the responsibility of the exiting process/thread to set the value of the `status_string_pointer` prior to being removed from the system. This will typically be performed in a process level exit handler or a last chance condition handler by issuing a call to `exec$set_exit_status_info()` specifying an item code of `e$c_status_string`.

6.7 Process/Thread Startup/Rundown Summary

This section is an attempt to summarize the steps that occur during the creation, execution, and termination of a thread in the Mica system. A very simple hierarchy will be studied in this description. The hierarchy consists of user.0, job.0, process.0, and thread.0 from Figure 6–1.
6.7.1 Startup Summary

The sample hierarchy is created as a result of the job controller calling exec$create_user(). The following steps occur as a result of this call.

1. A user object named user.0 is created. The user control block is initialized from the user_record parameter. The user object body is initialized to contain an empty job list and a job count of zero.

2. A job object named job.0 is created. The job control block is initialized by allocating q_per_job_limits quota from user.0, and assigning it to jcb_usage_and_limits. A job level container directory is created and optionally populated based on the existence of the job_initial_container parameter. The job object body is initialized to contain an empty process list and a process count of zero. The j_user_pointer is set to be a referenced pointer to user.0, and job.0 is linked to user.0’s job list. User.0’s job count is incremented to 1.

3. A process object named process.0 is created. The process control block is initialized by allocating q_per_process_limits quota from user.0 and assigning it to pcb_usage_and_limits. A security profile for the process is obtained from user.0. The accounting structure in the process control block is initialized to zero(). A process level container directory is created and optionally populated based on the existence of the process_public_container and process_private_container parameters. The pcb_condir_address and pcb_condir_mutex vectors are initialized. The process object body is initialized to contain an empty thread and sub-process list. The thread count and sub-process count is set to zero. The p_job_pointer is set to be a referenced pointer to job.0, and process.0 is linked to job.0’s process list. Job.0’s process count is incremented to 1.

4. A thread object named thread.0 is created. The thread control block is initialized by clearing all events and setting the tcb_irp_list_head to empty. The tcb_pcb_pointer field is initialized to point to process.0’s process control block. If specified in the thread_status parameter, the exit status object for the thread is referenced and stored in tcb_exit_status_ptr. The tcb_exit_status_value is cleared. The thread object body is initialized by setting the t_process_pointer to be a referenced pointer to process.0, and thread.0 is linked to process.0’s thread list. Process.0’s thread count is incremented to 1.

5. An address space is created for process.0 by calling e$create_process_address_space(). This call initializes portions of the tcb_thread_context, and the pcb_ptbr.

6. The kernel context for the thread is initialized by calling k$initialize_thread(). The thread is eligible to run in kernel mode at the e$initial_thread_startup() entry point. The creating thread calls e$release_temp_kernel_stack(). At this point, the original caller of exec$create_user() is returned to with a “successful” user creation. Failures in thread startup after this point occur in the context of the created thread and are treated as an abnormal termination status of the thread.

7. The first action performed by the thread at e$initial_thread_startup() is a call to e$initialize_address_space().

8. The process control region is allocated by calling e$allocate_pool() specifying a pool type of e$c_control_region_pool. The pcb_pcr_base field of process.0’s process control block is set to point to the allocated pcr, and the pcr is initialized by portions of the initial_thread_parameters parameter, and the object ID of process.0.

9. The thread control region is allocated by calling e$allocate_pool() specifying a pool type of e$c_control_region_pool. The tcb_tcr_base field of thread.0’s thread control block is set to point to the allocated tcr, and the tcr is initialized by portions of the initial_thread_parameters parameter, the object ID of thread.0, the address of process.0’s pcr, and various attributes of the thread specific address space.

10. The program image specified by the process_record field of the initial_thread_parameters parameter is mapped into process.0’s address space by transitioning into user-mode at e$program_image_startup().

6–46 Process Structure
11. Once at `program_image_startup()`, the thread issues a call to `exec$map_image()` and then sets the thread start address in the thread control region to the value returned by `exec$map_image()` by calling `exec$set_thread_information()`.

12. The thread entry point stored in `tcr_start_address` is "called" and is passed the thread parameters stored in `tp_thread_parameter_list`.

6.7.1.1 Additional Thread Startup Summary

This section describes the startup procedures for subsequent threads of a process. Assuming the hierarchy of the previous section, the following occurs when thread.0 makes a call to `exec$create_thread()` creating thread.1.

1. A thread object named thread.1 is created. The thread control block is initialized by clearing all events and setting the `tcb_irp_list_head` to empty. The `tcb_pcb_pointer` field is initialized to point to process.0's process control block. If specified in the `thread_status` parameter, the exit status object for the thread is referenced and stored in `tcb_exit_status_ptr`. The `tcb_exit_status_value` is cleared. The thread object body is initialized by setting the `t_process_pointer` to be a referenced pointer to process.0, and thread.1 is linked to process.0's thread list. Process.0's thread count is incremented to 2.

2. A partial address space is created for thread.1 calling `e$create_thread_stacks()`. This call initializes portions of the `tcb_thread_context`.

3. The thread control region is allocated by calling `e$allocate_pool()` specifying a pool type of `e$control_region_pool`. The `tcb_tcb_base` field of thread.1's thread control block is set to point to the allocated tcb, and the TCR is initialized by portions of the `initial_thread_parameters` parameter, the object ID of thread.1, the address of process.0's pcr, and various attributes of the thread specific address space.

4. The thread start address in thread.1's TCR is initialized to the value specified in the `thread_procedure` parameter.

5. The kernel context for the thread is initialized by calling `k$initialize_thread()`. The thread is eligible to run in kernel mode at the `e$subsequent_thread_startup()` entry point. At this point, the caller of `exec$create_thread()` is returned to with an "successful" thread creation. Failures in thread startup after this point occur in the context of the created thread and are treated as an abnormal termination status of the thread.

6. Once at `e$subsequent_thread_startup()`, the thread entry point stored in `tcr_start_address` is "called" and is passed the thread parameters stored in `thread_parameter_list`.

6.7.2 Rundown Summary

At some point in the threads lifetime it will either voluntarily exit by calling `exec$exit_thread()` or be forcibly exited by calling `exec$force_exit_thread()` on itself or having some other thread issue an `exec$force_exit_thread()` specifying that thread.

For the following rundown example, it is assumed that thread.99 issues an `exec$force_exit_thread()` specifying the object ID of thread.0. The hierarchy consists of user.0, job.0, process.0, and thread.0.

1. The Mica executive is entered at `e$force_exit_thread()`. The force-exit in progress flag is set in the thread object body of thread.0. The purpose of this flag is to prevent the creation of new exit handlers for the thread and to prohibit the thread from creating new threads, processes, and jobs.

2. The next step is to cause thread.0 to begin execution in "trusted" user-mode at the `e$in_context_force_exit()` executive entry point. At this point the force-exit of thread.0 is complete with respect
to thread.99. The following steps are then taken to force thread.0 into taking an active role in its exit. This occurs as follows:

- An elapsed timer is set to expire in a TBD period. If the timer expires, all of these steps are repeated, in addition to enabling user mode ASTs, setting the ast queue flush flag in the thread object body, and a call to k$flush_ast_queue() is issued.
- A user-mode AST is queued to thread.0. The target procedure of the AST is e$in_context_force_exit().

3. Once at e$in_context_force_exit(), thread.0 unwinds its stack by calling e$unwind().

4. Once the stack has been unwound and all unwind handlers have been executed, thread.0 executes a call to exec$exit_thread().

5. The code at exec$exit_thread() assigns the parameter exit_status to the tcb_exit_status_value field of thread.0's thread control block. If thread.0 was created with an exit status object, the exit_status is also assigned to tcb_exit_status_ptr^.es_exit_status_summary.status_value. The value exit_status is then assigned to pcb_exit_status_ptr^.es_exit_status_summary.status_value in process.0's exit status object. The force-exit in-progress flag for thread.0 is examined. Since the flag was set, the elapsed timer set up in e$force_exit_thread() is dismissed.

6. Thread.0 is then allowed to execute each one of its exit handlers. Since the thread is being force-exited, its exit handlers are assigned a small CPU time quota. When the quota expires, a user-mode AST is delivered to the thread that causes it to execute an exec$exit_thread(). The method for delivering the user-mode AST is similar to the technique used to cause the thread to execute at e$in_context_force_exit(), only the AST procedure target is e$exit_handler_expire(). The function of e$exit_handler_expire() is to simply call exec$exit_thread().

7. Thread.0 issues a call to e$remove_object_id() specifying its object id (t_obj_id). This action causes thread.0's object remove routine to be called.

8. Thread.0's object remove routine is entered. It performs the following steps.

   - All outstanding resources that require cleanup by the thread are processed. This includes the dismissal of all outstanding I/O by callin e$canceI_io_by_thread(), the dismissal of outstanding ASTs, and ...(TBS).
   - The thread control region is returned to the control region pool of process.0 by calling e$deallocate_pool().
   - The thread object is de-linked from the p_thread_queue_hd of process.0.
   - Since the above step causes the p_thread_count field to decrement to zero, the process.0 object is removed by calling e$remove_object_id() specifying the object ID of process.0.
   - If thread.0 was created with an exit status object, then the es_exit_status_event in the object is "set". The exit status object is then dereferenced by calling e$dereference_object().

9. The object remove routine for process.0 is entered as a result of thread.0's object remove routine being entered. The following occurs during process.0's object remove routine.

   - The job level container directory whose address is stored in pcb_condir_array is dereferenced.
   - The process level container directory is removed from the system by calling e$remove_object_id(), and specifying pcb_process_condir_id
   - The process control region is returned to its control region pool by calling e$deallocate_pool().

NOTE

If the timer in the example above had expired, that would indicate that the user-mode AST was not delivered, or that there was an exceptional delay in making progress through the stack unwind. In any case, timer expiration causes a retry which will eventually be successful.

6-48 Process Structure
• The process object is de-linked from the _process_queue_hd of job.0.
• Since the above step causes the _process_count field to decrement to zero, the job.0 object is removed by calling e$remove_object_id() specifying the object ID of job.0.
• The es_exit_status_event in process.0's exit status object is "set". The exit status object is then dereferenced by calling e$dereference_object().

10. The object remove routine for job.0 is entered as a result of process.0's object remove routine being entered. The following occurs during job.0's object remove routine.
   • The job level container directory is removed from the system by calling e$remove_object_id(), and specifying job_job_condir_id.
   • The job object is de-linked from the u_job_queue_hd of user.0.
   • Since the above step causes the u_job_count field to decrement to zero, the user.0 object is removed by calling e$remove_object_id() specifying the object ID of user.0.

11. The object remove routine for user.0 is entered as a result of job.0's object remove routine being entered. The routine performs no significant actions.

12. The original call in exec$exit_thread() which removed the object ID of thread.0 returns. The next step is a call to k$terminate_thread(). The purpose of k$terminate_thread() is to remove the specified thread (thread.0) from execution within the Mica system. Once all of the kernel related activities are complete, a pointer to thread.0 is queued to a special system thread known as the thread eater. The thread eater executes the loop at e$terminate_thread().

13. The function of e$terminate_thread() is to dequeue the thread's arriving on its queue, and to dereference the thread objects. When the thread eater processes thread.0, it calls e$dereference_object() specifying thread.0. The delete routine for thread.0 is entered. It is important to note that the delete routine for thread.0 is entered in the context of the thread eater.

14. The delete routine for thread.0 is entered. It performs the following actions.
   • Thread level accounting information is rolled up to the thread's process.
   • The thread specific address space (user-mode, and kernel-mode stacks) of thread.0 are returned to the address space of process.0 by calling e$delete_thread_stacks().
   • The referenced pointer to process.0 is dereferenced. This causes the delete routine for process.0 to be executed.

15. The delete routine for process.0 is entered. It performs the following actions.
   • An accounting record is written to the TBD message function processor. The information for the accounting record is obtained from the pcb_accounting field from process.0's process control block.
   • All resources accounted for in process.0's pcb_usage_and_limits are returned to job.0's jcb_usage_and_limits using the rules of deductable and non-deductable resource arithmetic.
   • The address space of process.0 is returned to the system by calling e(delete_process_address_space()).
   • The referenced pointer to job.0 is dereferenced. This causes the delete routine for job.0 to be executed.

16. The delete routine for job.0 is entered. It performs the following actions.
   • All resources accounted for in job.0's jcb_usage_and_limits are returned to user.0's ucb_quotas.q_usage_and_limits using the rules of deductable and non-deductable resource arithmetic.
   • The referenced pointer to user.0 is dereferenced. This causes the delete routine for user.0 to be executed.
17. The delete routine for user.0 is entered. It performs no significant actions.

18. Once the call frame has returned from the original call to $dereference_object$ issued by the thread eater on thread.0, the UJPT hierarchy consisting of user.0, job, process.0, and thread.0 is removed from the system, and the thread eater goes back to its queue of threads to be processed.

### 6.8 System Threads

This section describes the interface for creating system threads. It also describes the differences between system threads and normal threads, and the special restrictions placed on system threads.

#### 6.8.1 System Thread Creation

The $create_system_thread$ executive interface creates a system thread. The system thread executes within the UJPT hierarchy of the system. The address space of the system thread is that of the initial system process. Example 6–36 illustrates the interface to $create_system_thread$.

**Example 6–36: System Thread Creation Executive Interface**

```c
PROCEDURE exec$create_system_thread ( 
    OUT object_id : e$object_id; 
    IN object_parameters : exec$object_parameters = DEFAULT; 
    IN thread_procedure : e$thread_entry_point; 
    IN thread_record : e$thread_record = DEFAULT; 
    IN thread_immediate_parameter1 : e$thread_parameter = 0; 
    IN thread_immediate_parameter2 : e$thread_parameter = 0; 
    IN thread_in_current_address_space : boolean = false; 
    IN thread_status : e$object_id = DEFAULT; 
) RETURNS status; 
EXTERNAL; 
++
| Routine description: |
| Create a System thread object as specified by the parameters. |
| Arguments: |
| object_id - Object ID of the resulting process object |
| object_parameters - The object type independent parameters for the thread object |
| thread_record - Attributes of the thread being created |
| thread_immediate_parameter1 - Immediate parameter passed to thread through TCR |
| thread_immediate_parameter2 - Immediate parameter passed to thread through TCR |
| thread_procedure - pointer to thread entry point entry descriptor |
| thread_in_current_address_space - If true, the thread is created to run in the |
| callers address space which must be a system process; otherwise, |
| the thread is made to run in the address space of the initial system process |
| thread_status - Exit status object to be bound to the thread. If not present |
| then the thread is created without an exit status object |
| Return value: |
| TBS |
|--
```

#### 6.8.2 System Thread Restrictions

The important differences between system threads and normal threads are as follows:

- System threads may not execute in user-mode.
- System threads are incapable of processing or executing in the context of user-mode ASTs. Algorithms such as the one in $signal_thread$ that employ user-mode ASTs either understand system threads and modify their algorithms or don't support the functions on system threads.
- System threads execute within the address space of the system.

6–50 Process Structure
• There are no provisions for passing block data to a system thread through the `tcr_block_data` field in a system thread's TCR.
• The `exec$force_exit_thread()` system service is not supported for system threads.

6.9 System Processes

This section describes the interface for creating system processes. It also describes the differences between system processes and normal processes, and the special restrictions placed on system processes.

6.9.1 System Process Creation

The `e$create_system_process()` executive interface creates a system process. The system process executes within the UJPT hierarchy of the system as a sub-process. A new address space is created for the system process to execute within, but the process never executes in user mode. Example 6–37 illustrates the interface to `e$create_system_process()`.

Example 6–37: System Process Creation Executive Interface

```plaintext
PROCEDURE e$create_system_process ( 
    OUT object_id : exec$object_id; 
    IN object_parameters : exec$object_parameters = DEFAULT; 
    IN process_record : exec$process_record; 
    IN thread_object_parameters : exec$object_parameters = DEFAULT; 
    IN thread_procedure : e$thread_entry_point; 
    IN thread_record : exec$thread_record = DEFAULT; 
    IN thread_immediate_Parameter1 : exec$thread_parameter = 0; 
    IN thread_immediate_Parameter2 : exec$thread_parameter = 0; 
    IN thread_status : exec$object_id = DEFAULT; 
) 

// RETURNS status; 
EXTERNAL;
```

Routine description:

Create a Process and thread object as specified by the parameters.

Arguments:

- `object_id`: Object ID of the resulting process object
- `object_parameters`: The object type independent parameters for the process object
- `process_record`: Attributes of the process being created
- `thread_object_parameters`: The object type independent parameters for the thread object
- `thread_procedure`: Pointer to thread entry point entry descriptor
- `thread_record`: Attributes of the thread being created
- `thread_immediate_Parameter1`: Immediate parameter passed to thread through TCR
- `thread_immediate_Parameter2`: Immediate parameter passed to thread through TCR
- `thread_status`: Exit status object to be bound to the initial thread. If not present then the thread is created without an exit status object

Return value:

- `TBS`

6.9.2 System Process Restrictions

System processes can only execute system threads so all of the restrictions placed on system threads apply to system processes. The exception is that system processes execute within their own address space. They have the same object visibility as system threads for system and job level objects. Their process level object visibility is determined by the normal display container semantics described in Chapter 5, Object Architecture.
6.10 Mica Quota System Internals

This section briefly describes the interface to the Mica quota system. This section does not describe the block quota allocations that occur during job and process object creation, but focuses on the allocations that occur due to actions performed by a running thread.

6.10.1 Quota Levels

Quotas occur at three levels within the UJPT hierarchy.

- The User level.
- The Job level.
- The Process level.

At each level there is a data structure where the current quota usage is stored, and where the current quota limits are stored.

6.10.2 Quota Types

There are four types of quotas that are managed by the Mica quota system.

- Paging File quota.
- Paged Pool quota.
- Nonpaged Pool quota.
- Cpu Time quota.

The usage and limits stored at the various levels are actually vectors of integers indexed by quota type.

6.10.3 Quota Interfaces

The Mica system provides a "raw" quota interface for charging and returning quota, and an interface to the pool allocator the automatically charges quota for pool allocation and returns quota on pool deallocation.

6.10.3.1 Raw Quota Interface

The raw quota interface is used to charge and return quota at the specified level and of the specified type. Example 6–38 illustrates the interface to the raw quota system.
Example 6–38: Raw Quota Interface Definitions

PROCEDURE e$charge_quota {
    IN quota_type : e$quota_types;
    IN amount : integer;
    IN level : e$quota_level = e$f_process_quota_level;
};
EXTERNAL;

++

Routine description:

Charge quota to the current user, job, or process and determine if enough resources
are available to the object. If the appropriate limits field is -1 then no limit check
is performed. This routine acquires the per user, job, or process quota usage mutex which
is higher than the non-paged pool allocation mutex

Arguments:

quota_type - type of quota being charged (paged pool, paging file ...)
amount - the amount of quota requested
level - the level to charge the quota to (user, job, process)

Return value:

raises status$invalid_argument - if an invalid quota level is specified
raises status$quota_exceeded - the charge would put us over the limits for this level/type

--

PROCEDURE e$return_quota {
    IN quota_type : e$quota_types;
    IN amount : e$counter;
    IN level : e$quota_level = e$f_process_quota_level;
};
EXTERNAL;

++

Routine description:

Return quota to the current user, job, or process and determine if this causes an
underflow due to returning more than we have charged. This routine acquires the
per user, job, or process quota usage mutex which is higher than the non-paged pool
allocation mutex

Arguments:

quota_type - type of quota being returned (paged pool, paging file ...)
amount - the amount of quota being returned
level - the level to return the quota to (user, job, process)

Return value:

raises status$invalid_argument - if an invalid quota level is specified
raises status$quota_underflow - the return would cause the usage to become negative

--

6.10.3.2 Pool Quota Interface

There is a front end interface on the pool allocator which can allocate pool and charge quota in a
single call. An interface also exists to deallocate and return quota on a single call. Most users of pool
will use the charging interface. The exception is the object architecture which uses the raw interface,
since the quota charge is performed when the object is inserted into a container rather then when
the pool is allocated.
Example 6–39 illustrates the interface to the pool allocator for charging and returning quota.

**Example 6–39: Pool Allocator with Quotas Interface**

```haskell
PROCEDURE `allocate_pool_and_quota` (  
  IN pool_type : epool_index;  
  IN number_of_bytes : Integer;  
  IN wait_for_free : boolean;  
  IN quota_level : equota_level = eeq_process_quota_level;  
  OUT starting_block_address : POINTER anytype;  
) RETURNS Integer;  
! FIXFIX status  
EXTERNAL;

ROUTE DESCRIPTION:
allocate memory from the specified pool and charges quota.
Paged pool allocation is charged to paged_pool_quota, all others are charged to nonpaged_pool_quota.

ARGUMENTS:

pool_type - the type of pool allocation is to be made from
number_of_bytes - the minimum number of bytes required to satisfy the request
wait_for_free - if no pool is available and this flag is set, then thread will enter into a resource wait
quota_level - the level to charge quota to (defaults to process level)
starting_block_address - the address of the allocated pool or NIL

RETURN VALUE:

raises status_no_free_pool = pool exhausted and wait_for_free is set to false
raises status_quota_exceeded = not enough quota exists to satisfy allocation

ROUTE DESCRIPTION:
return the block of pool pointed to by starting_block_address to the proper pool and charge quotas.
Paged pool deallocation is returned to paged_pool_quota, all others are returned to nonpaged_pool_quota.

ARGUMENTS:

starting_block_address - pointer to a block of pool
quota_level - the level to charge quota to (defaults to process level)

RETURN VALUE:

raises status_quota_underflow - if pool that was not charged for is being returned, then it will be caught here
```

6-54 Process Structure
CHAPTER 7
MEMORY MANAGEMENT

7.1 Overview

7.1.1 Requirements

The memory management subsystem provides a combination of hardware and software functions to accomplish the mapping of physical address space into the virtual address space of a process. The physical address is used by hardware to identify a page in physical memory. The memory management subsystem has six principal requirements:

- A number of processes may occupy main memory simultaneously, all freely using their own unique address spaces, while only accessing their own data and code.
- Only a portion of the total address space for a process needs to be resident at any one time.
- The data and code belonging to a process are scattered throughout physical memory and need not be contiguous.
- Processes can automatically share code and data.
- Processes are protected from themselves and from other processes.
- Support for the I/O system. This includes mapping of I/O space, and locking pages in memory for I/O.

7.1.2 Functional Description

The Mica memory management is designed to support a large user virtual address space (2 gigabytes per address space), and large working sets (4 gigabytes per address space). Figure 7-1 illustrates the layout of the virtual address space associated with a process.

7.1.2.1 Environment of Memory Management

The memory management subsystem executes in kernel mode. Through the use of mutexes, multiple processors may be executing within the memory management subsystem simultaneously. During handling of the translation not valid fault, ATs are disabled. This prevents an AST from interrupting the translation not valid fault processing, causing a recursive entry into the translation not valid code.

The memory management subsystem consists of the following features.

- Fault handlers for access violation — Checks to see if the offending page is a guard page for a user's stack. If so, the guard page is unprotected and a condition indicating the stack guard page was accessed is raised. Otherwise, an access violation condition is raised.
- Fault on read—Raises an access violation condition.
- Fault on write—Implements copy-on-modify semantics, and helps to track the modified state of a page.
• *Fault on execute*—Checks to see if the page is a kernel entry page for system service dispatching. If so, it saves appropriate registers, and calls the system service. Otherwise, it raises an access violation condition indicating that the user attempted to execute nonexecutable data.

• *Translation not valid*—Implements the pager. The page table entry for the faulting page is examined to determine how to make the page valid. The faulting page can come from a mapped file, a paging file, a page of zeroes, or a page that is already in memory. In the latter case, the page that is already in memory is either in a transition state, or shared with another address space that already has the page valid. This is also referred to as a *page fault*.

In addition, there are a number of system routines that contribute to memory management. These include:

• System services—Affect an address space

• Executive services—Manage and allocate pages from paged and nonpaged pools; also probe, lock, and unlock I/O buffers from memory

• Balance set manager—Ensures ample free pages

• Modified page writer—Writes modified pages

7.1.3 Memory Management Data Structures

The following system data structures are used by the memory management subsystem:

• *Page frame number (PFN) database*—Tracks physical pages and their states. Each physical page is in one of five states:
  1. Active and valid—A page in this state is mapped in some address space’s working set.
  2. Free—Available for immediate reuse.
  4. Standby—A page in this state is marked as in transition in a (prototype) page table entry (PTE), and may be reactivated as the result of a page fault for the transition page. This page can be reused, but the page table entry must change from a transition state to an invalid state.
  5. Modified—A page in this state is marked as in transition in a (prototype) PTE, and may be reactivated as the result of a page fault for the transition page. Before the page can be reused its contents must be written to disk. Once its contents are written to disk, the page enters into the standby state.

• Working set list—Manages the physical pages owned by an address space. Each address space is guaranteed a certain number of physical pages. When a page fault occurs which would cause that number of pages to be exceeded, a physical page is removed from that address space by making the page table entry invalid and decrementing a usage count for the page. The working set list is a list of virtual page numbers which are currently valid.

• Page table pages—Manage the complete address space. Each address space contains one segment 1 page table page which is one page in size and contains 512 page table entries. There are multiple, up to 512, segment 2 page table pages which are one page in size and contain 1024 page table entries.

Each page table entry is a quadword and indicates whether a page exists at the corresponding address, and whether the page is valid or not. If a page is not valid, its page table entry indicates how to make the page valid.

7–2 Memory Management
Mapping objects, section objects and segment objects—Track mapped files.

Figure 7-1: Virtual Address Space Layout

<table>
<thead>
<tr>
<th>Memory Region</th>
<th>Size</th>
<th>Base Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>No Access — 64 KB</td>
<td>00000000</td>
</tr>
<tr>
<td>User Space — 2 GB (less 64 KB)</td>
<td></td>
<td>00010000</td>
</tr>
<tr>
<td>(All pages owned by user.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Kernel access and user access</td>
<td></td>
<td></td>
</tr>
<tr>
<td>are always identical.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2 GB</td>
<td>Shareable Image Space — 0.5 GB</td>
<td>80000000</td>
</tr>
<tr>
<td>(All pages owned by user.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Kernel access and user access</td>
<td></td>
<td></td>
</tr>
<tr>
<td>are always identical.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2.5 GB</td>
<td>Control Space — 64 MB</td>
<td>A0000000</td>
</tr>
<tr>
<td>(Owned by kernel.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4 GB</td>
<td>System Space — 1.5 GB less (64 MB + 8 MB)</td>
<td>A4000000</td>
</tr>
<tr>
<td></td>
<td>Paged System Area</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Nonpaged System Area</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Hyper Space — 4 MB</td>
<td>FF8000000</td>
</tr>
<tr>
<td></td>
<td>Hyper Space Working Set Lists — 4 MB</td>
<td>FFC000000</td>
</tr>
</tbody>
</table>

Notes on Figure 7-1:

- **User space**—Maps user code and data. Includes 64 KB that is set no access to catch programming errors.
- **Shareable image space**—Maps system-wide installed shareable images.
- **Control space**—Maps kernel-mode stacks and other thread-related structures.
- **System space**: Paged area—Maps pages that can be paged to disk. This area includes code, data, and pool.
- System space: Nonpaged area—Maps pages that must be memory resident. This area includes code, data, and pool.
- **Hyper space**—Maps address space page tables and data structures.
7.1.4 Differences from the VAX/VMS Memory Management Subsystem

Mica memory management supports several enhancements over VMS memory management, including the following:

- An address space's page table pages are only valid within that address space.
- Image files are automatically shared among all address spaces executing the image file.
- Copy-on-modify operations.
- Large and sparse address spaces.
- Standby and zeroed page lists.
- No system working set. Each address space's working set contains the portion of the system that can bepaged that is used by that address space.
- No swapper. Reduction of address space use is accomplished by paging the process out of memory.
- Virtual addresses cannot be overmapped, without first deleting the previous virtual addresses.

7.2 Introduction

This chapter describes the memory management subsystem used by the Mica operating system. This subsystem includes the memory management hardware functions and the software that handles memory management faults. Presently, this chapter only deals with the 32-bit version of the PRISM architecture, and therefore refers to page size as 8 Kbytes and makes all calculations based on 8K pages. The memory management system, however, is designed to allow page sizes up to 64 Kbytes. To support the larger page size, all user-visible attributes fall on 64-Kbyte boundaries.

The memory management subsystem allows a combination of hardware and software functions to accomplish the mapping of physical address space into a process's virtual address space. The software portion of memory management is responsible for initializing the page table entries (PTEs) for each address space, managing physical memory, handling faults (translation not valid faults, access violation faults, fault on read, fault on write, and fault on execute faults) handling each process's working set, and handling the migration of pages in and out of physical memory. Additionally, memory management implements change mode dispatching of system services.

7.2.1 Virtual Address Space

A virtual address is a 32-bit unsigned integer that specifies a byte location within the virtual address space. The program sees a linear array of 4,294,967,296 bytes (4 gigabytes). The virtual address space is divided into pages, which are the units of relocation, sharing, and protection. Each page is 8,192 bytes in length (8 Kbytes). Hence, the 4-gigabyte virtual address space consists of 524,288 virtual pages.

7.2.1.1 Virtual Address Format

The PRISM processor generates a 32-bit virtual address for each instruction and operand in memory. The virtual address consists of two segment number fields, and a BYTE_WITHIN_PAGE field.

Figure 7–2 shows the format for a virtual address.

Figure 7–2: Virtual Address Format

<table>
<thead>
<tr>
<th>31</th>
<th>23</th>
<th>12</th>
</tr>
</thead>
<tbody>
<tr>
<td>SEG1_NUMBER (9 bits)</td>
<td>SEG2_NUMBER (10 bits)</td>
<td>BYTE_WITHIN_PAGE (13 bits)</td>
</tr>
</tbody>
</table>

7–4 Memory Management
The segment number fields, bits <31:13> of a virtual address, specify the virtual page to be referenced. The BYTE_WITHIN_PAGE field, bits <12:0> of a virtual address, specifies the byte offset within the page.

### 7.2.2 Physical Address Space

Physical addresses are, at most, 45 bits. A processor may choose to implement a smaller physical address space by not implementing some of the high-order bits. The most significant implemented physical address bit selects memory space when it is 0, and I/O space when it is 1.

### 7.2.3 Page Table Entries

The processor uses quadword page table entries (PTEs) to translate virtual addresses to physical addresses. There is a PTE for each page in virtual address space. A PTE contains hardware and software control information and, when valid, the physical page frame number (PFN).

Figure 7–3 shows the format of a PTE as defined by the PRISM architecture.

#### Figure 7–3: Page Table Entry

<table>
<thead>
<tr>
<th>31</th>
<th>1</th>
<th>1</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>PFN&lt;18:0&gt;</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Reserved</td>
<td>AS</td>
<td>SF</td>
<td>U</td>
<td>W</td>
<td>R</td>
<td>E</td>
<td>V</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Reserved for Software</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Fields in the page table entry are interpreted as follows:
<table>
<thead>
<tr>
<th>Bits</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Valid (V)—Indicates the validity of the ASM, FOE, FOW, FOR bits and the PFN field. When V is</td>
</tr>
<tr>
<td></td>
<td>set, the ASM, FOE, FOW, FOR bits and the PFN fields are valid for use by hardware. When V is</td>
</tr>
<tr>
<td></td>
<td>clear, the PFN field is reserved for use by software. The V bit does not affect the validity of</td>
</tr>
<tr>
<td></td>
<td>KWE, URE, and UWE.</td>
</tr>
<tr>
<td>1</td>
<td>Kernel Read Enable (KRE) — This bit enable reads from kernel mode. If this bit is a 0 and a LOAD</td>
</tr>
<tr>
<td></td>
<td>or instruction fetch is attempted while in kernel mode, an access violation occurs. This bit is</td>
</tr>
<tr>
<td></td>
<td>valid even when V=0.</td>
</tr>
<tr>
<td>2</td>
<td>Kernel Write Enable (KWE) — This bit enables writes from kernel mode. If this bit is a 0 and a</td>
</tr>
<tr>
<td></td>
<td>STORE is attempted while in kernel mode, an access violation occurs. This bit is valid even when</td>
</tr>
<tr>
<td></td>
<td>V=0.</td>
</tr>
<tr>
<td>3</td>
<td>User Read Enable (URE) — This bit enables reads from user mode. If this bit is a 0 and a LOAD</td>
</tr>
<tr>
<td></td>
<td>or instruction fetch is attempted while in user mode, an access violation occurs. This bit is</td>
</tr>
<tr>
<td></td>
<td>valid even when V=0.</td>
</tr>
<tr>
<td>4</td>
<td>User Write Enable (UWE) — This bit enables writes from user mode. If this bit is a 0 and a STORE</td>
</tr>
<tr>
<td></td>
<td>is attempted while in user mode, an access violation occurs. This bit is valid even when V=0.</td>
</tr>
<tr>
<td>5</td>
<td>Fault On Read (FOR) — When set, a fault on read exception occurs on an attempt to read any</td>
</tr>
<tr>
<td></td>
<td>location in the page.</td>
</tr>
<tr>
<td>6</td>
<td>Fault On Write (FOW) — When set, a fault on write exception occurs on an attempt to write any</td>
</tr>
<tr>
<td></td>
<td>location in the page.</td>
</tr>
<tr>
<td>7</td>
<td>Fault On Execute (FOE) — When set, a fault on execute exception occurs on an attempt to execute</td>
</tr>
<tr>
<td></td>
<td>an instruction in the page.</td>
</tr>
<tr>
<td>8</td>
<td>Address Space Match (ASM) — When set, this PTE matches all address space numbers.</td>
</tr>
<tr>
<td>12:9</td>
<td>Reserved for future use by DIGITAL.</td>
</tr>
<tr>
<td>44:13</td>
<td>Page Frame Number (PFN) — The PFN field always points to a page boundary. If V is set, the PFN</td>
</tr>
<tr>
<td></td>
<td>is concatenated with the BYTE_WITHIN_PAGE bits of the virtual address to obtain the physical</td>
</tr>
<tr>
<td></td>
<td>address. If V is clear, this field may be used by software.</td>
</tr>
</tbody>
</table>

### 7.2.4 Software Flags Within the PTE

Figure 7-4 shows the format for an invalid PTE (Invalid PTE's have a VALID bit set to 0.)

**Figure 7-4:** Invalid Page Table Entry

![Diagram of PTE Format](image)

The following bits reside in the reserved for software portion of the PTE for use by the memory management subsystem:

7–6 Memory Management
Bits | Description
---|---
63 | Kernel Entry Page (KEP)—The page is used as an entry point to system service dispatching. FOE is set, and the page protection is set as kernel write, kernel read, user read, and owned by kernel mode.
62 | Copy On Modify (COM)—FOW is set. When an FOW fault occurs for the page, the pager provides the address space a private copy of the page to modify. (See Section 7.8.4.6 for more information.)
61 | Section Member (SEC)—The page was created as the result of a Map Section service call. It cannot be deleted via a user mode call to exec$delete_address_space.
60 | Owner (OWN)—Indicates the owner of the page; a 1 represents user mode, a 0 represents kernel mode. If a page is owned by user mode, the kernel and user protection must always be identical. For example, if the user protection is read-only, then the kernel protection must be read-only as well. Ownership of all virtual pages is static as shown in figure Figure 7–7.
59 | Prototype PTE (PRO)—This PTE refers to a prototype PTE. A prototype PTE has the same format as a PTE and is used to facilitate sharing of pages. (See Section 7.7.4 for more information.)
58 | Demand Zero (DZO)—Creates a page of zeros upon reference to the page.
57 | Transition Page (TRN)—The page is in a transition state and is currently on the standby list or modify list, or a page in the process of being read.
56 | Shareable (SHR)—The page may be shared among all users.
55 | Page File Offset Format (PFO)—The FRAME field of the PTE is in page file offset format.
54 | Guard Page (GRD)—The page is a stack guard page. When the page is accessed, create a page and raise a "guard page referenced" exception.
53 | Start of allocation (SOA)—The page is the first page of a system pool allocation.
52 | End of allocation (EOA)—The page is the last page of a system pool allocation.
45:51 | Cluster Factor—A number between 0 and 127.

Figure 7–5 shows the format for a valid PTE.

**Figure 7–5: Valid Page Table Entry**

<table>
<thead>
<tr>
<th>Page Frame Number &lt;18:0&gt;</th>
<th>Reserved</th>
<th>A</th>
</tr>
</thead>
<tbody>
<tr>
<td>K</td>
<td>C</td>
<td>S</td>
</tr>
<tr>
<td>63</td>
<td>62</td>
<td>61</td>
</tr>
</tbody>
</table>

The following flags are within the reserved for software portion of the PTE for use by the memory management subsystem.
7.2.4.1 Pillar Record Definition for PTE

```
mm$pte : RECORD
  valid : bit;
  kernel_read_enable : bit;
  kernel_write_enable : bit;
  user_read_enable : bit;
  user_write_enable : bit;
  fault_on_read : bit;
  fault_on_write : bit;
  fault_on_execute : bit;
  address_space_match : bit;
  hw0 : bit;
  hw1 : bit;
  hw2 : bit;
  hw3 : bit;
  UNION CASE *
  WHEN 1 THEN
    page_frame_number : integer [..] SIZE(bit,32);
    working_set_index : integer [0..2**14-1] SIZE(bit,14);
  WHEN 2 THEN
    UNION CASE *
      WHEN 1 THEN
        page_file_offset : integer [0..2**19-1] SIZE(bit,19);
        page_file_index : integer [0..2**13-1] SIZE(bit,13);
      WHEN 2 THEN
        prototype_pte_address : integer [..] SIZE(bit,32);
      END UNION;
    cluster_factor : integer [0..2**7-1] SIZE(bit,7);
    end_of_allocation : bit;
    start_of_allocation : bit;
    stack_guard : bit;
    page_file_format : bit;
    share : bit;
    transition : bit;
    demand_zero : bit;
  END UNION;
  prototype_pte : bit;
  owner : mm$mode_packed_type;
  section : bit;
  copy_on_modify : bit;
  kernel_entry_page : bit;
END RECORD;
```

7–8 Memory Management
7.2.5 Address Translation (from the PRISM SRM)

Address translation is performed by accessing entries in a two-level page table structure. The page table base register (PTBR) contains the physical page frame number of the first-level page table. If part of any page table resides in /O space, or in nonexistent memory, the operation of the processor is UNDEFINED.

The page table base register contains the physical page frame number of the highest-level page table, which is the segment 1 page table. Bits <31:23> of the virtual address are used to index into the first-level page table to obtain the physical page frame number of the base of the second-level page table, which is the segment 2 page table. Bits <22:13> of the virtual address are used to index into the second-level page table to obtain the physical page frame number (PFN) of the page being referenced. The PFN is concatenated with virtual address bits <12:0> to obtain the physical address of the location being accessed.

If the first-level PTE is valid, the protection bits are ignored; the protection code in the second-level PTE is used to determine accessibility. If a first-level PTE is invalid, an access violation occurs if the PTE<KE> equals zero. An access violation on a first-level PTE implies that all lower-level page tables mapped by that PTE do not exist.

Figure 7-6 depicts the steps taken to translate a valid virtual address to a physical address.

7.3 Entering Memory Management

Memory management-related faults trigger the invocation of memory management software. Five types of faults are handled in the memory management software: access violation, translation not valid, fault on write, fault on read, and fault on execute.

An access violation fault occurs if, during a virtual address translation, the segment 2 PTE protection field indicates that the current process does not have access to the data. Additionally, an access violation fault occurs if the thread attempts to access a virtual address that translates to a nonvalid segment 1 PTE with kernel read access disabled, indicating that no segment 2 PTEs exist for this segment. All access violation faults are examined to see if the thread has exhausted the stack and is attempting to access the "guard" page. If so, memory management software changes the protection on the "guard" page to allow user access, creates a page of zeroes to map the guard page, and raises a guard page referenced exception.

A translation not valid fault occurs if, during the virtual address translation, the VALID bit is 0 in either the segment 1 PTE or the segment 2 PTE, and access is allowed. Note that access violation checks are performed before translation not valid. This condition is referred to as a page fault, and the memory management subsystem is invoked to get the proper page into physical memory.

A fault on read fault occurs when a virtual address translation encounters a valid segment 2 PTE with the FOR bit set and the access type is read. Fault on read is used to implement execute-only protection, which is described later.

A fault on write fault occurs when a virtual address translation encounters a valid segment 2 PTE with the FOW bit set and the access type is write. The memory management subsystem examines the software flags in the PTE and takes the necessary action to resume the user's program. Fault on write is used on all writable pages to track whether the page was actually modified by the user. A modified page must be saved on disk if it is removed from physical memory.

Fault on write is also used to implement copy on modify. When an FOW fault occurs for a copy-on-modify (COM) page, the pager creates a private copy of the page for the user to modify. This is accomplished by allocating a physical page, copying the contents of the page on which the fault occurred to the allocated physical page, and inserting the newly copied page into the address space's page table, replacing the previous shared page. The copied page is now a private page with all ties to the prototype PTE eliminated. Optimizations are performed during this operation; for example, if the shared COM page has only one address space referencing it, it can be turned into a private page, and the next reference to the shared page results in the page being read into memory.
Copy on modify is a key concept in the PRISM memory management structure. It conserves physical memory and facilitates sharing. It allows any read/write page to be shared in a read-only mode. When an attempt is made to write into the page, memory management software creates a private copy of that page. The original copy is no longer shared. It also allows shared execute-only code to be modified for debugging purposes. For example, a breakpoint can be placed into a shared run-time library. When the modification is made to the shared page, an FOW fault occurs and the pager, finding the COM flag set, creates a private page for subsequent writes by the debugger.
A fault on execute fault occurs when a virtual address translation encounters a valid segment 2 PTE with the FOE bit set and the access type is execute. The memory management subsystem examines the software flags in the PTE and takes the necessary action to resume the user's program. Fault on execute is used to implement change mode dispatching to system services and nonexecute protection on data.

7.3.1 Execute-Only Protection

The PRISM hardware does not support execute-only protection. However, the memory management software supports execute-only protection by setting fault on read in the PTE for pages which are to be execute-only. Additionally, the page is protected as read-only, disallowing write access to the page. When a fault on read fault occurs on a page, memory management software raises an access violation.

Note that the PRISM probe read instruction does not check the fault on read bit. If system software probes an address for read access, it may get an access violation when reading that address and must have a condition handler established for this condition.

By use of ACLs, a file may be protected such that a user is not allowed to change the protection of a page. This provides some protection from users attempting to decode proprietary software. More information protecting files using ACLs can be found in Chapter 10, Security and Privileges.

The current design only checks ACLs when the page being changed refers to a prototype PTE. Any protection changes to private pages (even those pages which were made private by a copy-on-modify operation) always succeeds. The change of protection would fail if the user did not have read access to the image.

\ How does a file ACL extend to a segment ACL? \n
7.3.2 Nonexecute Protection

Nonexecute protection is accomplished by setting the protection of the page appropriately (read-only or read/write and setting the fault on execute (FOE) bit in the PTE). Any attempt to execute data within the page causes the fault on execute handler to be entered, which upon analyzing the PTE, raises an access violation.

7.4 Layout of Virtual Address Space

The 4-gigabyte virtual address space is partitioned into five parts:

1. User space—Consists of 2 gigabytes, which is unique for each address space on the system (though cooperating processes may share code and data).

2. Shareable image space—Consists of a ¼ gigabyte, which maps system-wide installed shareable images. Shareable images residing in this space become "based" images, and all address fixups are performed when installed. When a thread references a shareable image in this space, it is mapped into the address space at the base address at which the shareable image was installed.

3. Control space—Consists of 64 megabytes, which is unique for each address space.

4. System space—Consists of almost 1¼ gigabytes, which is shared among all processes.

5. Hyper space—Consists of 8 megabytes. The last PTE in the segment 1 page table contains the value of the PTBR, allowing each process to map its own page table entries to the same location. This is unique for each process.

Figure 7-7 shows the layout for the virtual address space.
Figure 7-7: Layout of Virtual Address Space

<table>
<thead>
<tr>
<th>Address Range</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 - 64 KB</td>
<td>No Access</td>
</tr>
<tr>
<td>64 KB - 2 GB</td>
<td>User Space</td>
</tr>
<tr>
<td>2 GB - 0.5 GB</td>
<td>Shareable Image Space</td>
</tr>
<tr>
<td>2.5 GB - 64 MB</td>
<td>Control Space</td>
</tr>
<tr>
<td>1.5 GB - 64 MB</td>
<td>System Space</td>
</tr>
<tr>
<td>4 MB</td>
<td>Hyper Space</td>
</tr>
<tr>
<td>4 MB</td>
<td>Hyper Space Working Set Lists</td>
</tr>
</tbody>
</table>

### 7.4.1 User Space

User space is defined as the address range from 0 to 7FFFFFFF, inclusive. The first 64 Kbytes is defined as no access as an aid in catching programming errors. All pages in user space are owned by user mode and the protection of each page for user and kernel is identical. If the protection on a page is changed by the Set Page Protection executive service, the kernel protection changes as well. Protection on a page cannot be changed unless the page is a private page, or if the ACL on the segment object containing the page grants the user the rights to change the protection on the page.

By guaranteeing that the kernel and the user have identical protection on a page, the user is prevented from issuing an executive service, and then from another thread of execution within the same address space, changing the protection of the page. This could compromise the integrity of the system. By allowing no kernel-owned pages in user space, and by always keeping user and kernel access identical, the worst that can happen is that the executive service gets an access violation when accessing an address which previously passed probing tests. System services enable exception handlers, and return the error condition to the user through the status parameter.
7.4.2 Shareable Image Space

The shareable image space, defined by the address range 80000000 to 9FFFFFFF, is reserved for installed images that reside at constant addresses in each address space. This allows frequently used shareable images to reside at a constant address. This minimizes activation time since prototype PTEs and some address fixups exist for these images. The INSTALL/BASE command installs images in the shareable image space. All local address fixups are done on the image. When an image is run that is linked to a shareable image resident in the shareable image space, the image fixups are performed to map the shareable image.

All pages in the shareable image space follow the same ownership and protection rules as pages in user space. This prevents a user from changing the protection of a shared page to user write, issuing a executive service that would write into the page, then (through another thread), deleting the virtual address containing the page and remapping the virtual address, having the page appear as read-only. Since the kernel protection is identical to the user protection, an access violation would occur, and an error status would be returned from the executive service.

7.4.3 Control Space

The control space is a 64-megabyte area from A0000000 to A3FFFFFF, inclusive. It is reserved for thread-specific structures such as kernel stacks. Each address space has a unique control space. All pages are owned by kernel mode.

At process creation, an area within the control region is allocated as the process control region. The process control region's page protection allows user reads, and kernel writes. The process control region contains information about the status of a process such as current number of executing threads. This page is locked in the working set.

At thread creation, an area within the control region is allocated as the thread control region. The protection on this area allows user reads, and kernel writes. The thread control page contains information about the thread it represents, such as the user stack's base and bounds. Note that the stack range does not include the guard page(s). When the guard pages are made valid, the stack limit value is changed to reflect the new end of the stack.

At thread creation, a kernel stack is allocated. The protection on the kernel stack allows kernel writes, but prohibits user access. The kernel stack is locked in the working set.

If a thread within the process is using vector instructions, the thread's vector save page is allocated from the control region and locked in the process's working set.

7.4.4 System Space

System address space is the range of addresses between A4000000 and FF7FFFFFFF, inclusive. It is imperative that each address space has the same system address space mapping. This allows all executive services to call the operating system at the same address, and allows interrupts to be serviced in the context of any process. All pages in system space are owned by the kernel.

7.4.4.1 Kernel Mode Entry Pages

Kernel mode entry pages reside within system space and are protected to allow user reads, kernel reads, kernel writes, and faults on execute. Because the pages are owned by kernel mode, the user is unable to delete the pages, change the protection, or write into the page.
7.4.4.2 Nonpaged System Space

At system initialization, the system address space is divided into two parts: the nonpagable system and the pagable system. The nonpagable system consists of portions of the executive which must be memory resident (for example, memory management, the PFN database, certain drivers, the low-level kernel, the scheduler, and nonpaged pool).

The nonpagable system space is created by preallocating, at system initialization, the maximum number of segment 1 page table entries required by the nonpaged portion of the executive. Each segment 1 page table entry reserved allows eight megabytes of nonpaged memory for the executive. Each process address space has the same nonpagable system space segment 1 page table entries in the same location. This ensures that all address spaces have the same view of the nonpaged portion of the executive, even if changes are made in the segment 2 page table pages for the nonpaged executive. As the memory requirements for the nonpaged portion of the executive increase, the nonpagable system’s segment 2 page table entries become valid, and all processes automatically map the current state of the nonpagable system.

Nonpaged system space may be deleted by the executive by use of the Delete Virtual Address Space service. This allows the executive to free pages from nonpaged pool when they are no longer referenced.

At system initialization, all nonpaged pool is virtually and physically contiguous. For nonpaged pool less than a some determined size, the physical pages are all in the lower ½ gigabyte of memory. This allows device function processors to allocate nonpaged pool for controller packets from nonpaged pool, without having to worry about physical addresses being in the first ½ gigabyte.

7.4.4.3 Paged System Space

The paged portion of the system consists of shared segment objects which are mapped into the same virtual address space for all processes on the system. At address space creation, each segment 1 PTE of paged system address is invalid and contains the system virtual address of the prototype page table which maps the segment 2 page table pages. The prototype page table contains one entry for every eight megabytes of paged system space, and is guaranteed to be valid because it resides in nonpaged system space. This allows each process’s working set to contain only those pages of the pagable system currently being utilized by that process. Figure 7–8 shows the layout of the system space at the time of address space creation.

7.4.5 Hyper Space (Outer Space)

The last eight megabytes of the virtual address space are reserved for mapping the address space page table pages and the address space data structures. It is unique for each address space. All PTEs in the segment 1 page table are owned by kernel mode and are protected to prevent access from user mode. This prevents the user from examining page tables.

Each process’s page table pages are mapped at the same address, and are not in system space. This means that other process’s address space cannot be referenced from the current process, and system threads and device drivers must be aware of this fact. A side effect is that there is no system generation parameter to control the maximum number of virtual pages that can be mapped by a process, nor is there one for the maximum working set. Both these resources are managed by user quotas and limits imposed by physical memory constraints.

The segment 1 PTEs only occupy 4 Kbytes of the page, the second 4 Kbytes are used as segment 2 PTEs to map the address space tables (for example, the working set list) for the process. The 4-megabyte address range of FFC00000 to FFFFFFFF contains the address space tables.
7.4.6 Locating the PTE for a Virtual Address

When a translation not valid fault is raised, the memory management subsystem is entered in kernel mode. Figure 7–9 shows the information contained on the kernel stack.
The memory management subsystem uses the `RELATED_VIRTUAL_ADDRESS_IN_PAGE` field, which is on the top of the stack, to find the invalid PTE.

### 7.4.6.1 Locating the Segment 1 PTE Address
Let RVA represent the `RELATED_VIRTUAL_ADDRESS_IN_PAGE` field found on the top of the stack. Bits `<31:23>` of the RVA are the index into the segment 1 page table. The virtual address of the segment 1 PTE field for the RVA is `FFBFE000 + RVA<31:23> * 8`. Expressed in the form (seg1, seg2, offset), this is (1FF, 1FF, RVA<31:23> * 8). Note that in this case, the segment 1 PTE is being used as a segment 2 PTE as well.

### 7.4.6.2 Locating the Segment 2 PTE Address
The virtual address of the segment 2 PTE field for the RVA is `FF800000 + RVA<31:13> * 8`. Expressed in the form (seg1, seg2, offset), this is (1FF, RVA<31:23>, RVA<22:13> * 8).

### 7.4.6.3 Translating a Virtual Address to Physical Address
I/O function routines executing in the process's context need to convert a valid virtual address to a physical address. The page frame number, bits `<44:13>` of a PTE, can be located by finding the segment 2 PTE, as above, and extracting the page frame number. The physical address is the page frame number in bits `<44:13>` and the `BYTE_WITHIN_PAGE` field in bits `<12:0>`.

### 7.4.6.4 Locking Pages in Memory for I/O
The memory management subsystem has procedures that are invoked by I/O routines to lock pages in memory during I/O. The `e$probe_and_lock_buffer` routine has the following interface:

```plaintext
PROCEDURE e$probe_and_lock_buffer (
    IN host_transfer_list : POINTER e$host_transfer_list;
    IN access_mode : k$processor_mode; !k$co_user or k$co_kernel
    IN operation_type : e$io_operation; ! (e$c_read_into_buffer,
        ! e$c_write_from_buffer,
        ! e$c_modify_buffer)
)
```
Routine description:

This routine probes the virtual address range for the desired access and locks the buffers in memory. If a virtual address is not valid the page is made valid by either faulting it in or in the case of e$C_read_into_buf for a full page, demand zeroed.

Arguments:

IN host_transfer_list - pointer to a host transfer list. From the host transfer list the starting virtual address and length can be obtained. This routine completes the host transfer list by filling in the process_control_block address, the offset field, and the page_frame_number array.

IN access_mode - mode in which to do the probes on the buffer

IN operation_type - type of operation, read, write or modify

Return value:

none - any errors raise a condition. And the PFN elements of the host transfer list are set.

--

If the operation_type indicates e$C_read_into_buffer (that is, read into memory causing a memory write) then the MODIFY bit in the PFN database is set for each page that is locked for I/O. This is required because a DMA write into memory will not cause an FOW fault or other event that would be noticed by the memory management software. If the page is copy-on-modify (COM bit is set), then the copy-on-modify actions are performed at this time.

The host transfer list, described in the Chapter 8, I/O Architecture, contains the starting virtual address and the length of the buffer.

7.4.6.4.1 Steps to Lock Pages in Memory

Memory management software uses the following steps to lock pages into memory:

1. Ensures that the dynamic portion of the working set is adequate for the number of pages which need to be locked and the current number of pages locked for I/O by the process plus this locking request does not exceed the dynamic portion of the working set

2. Finds the PTE for the page to be locked for I/O

3. Checks the PTE access fields to ensure the access mode and operation is compatible with the PTE

4._faults the page into the working set if the page is not valid (Note, the pager performs an optimization at this point. If a whole page is going to be overwritten, the pager substitutes a page of zeroes for the real page to avoid doing a read from the disk. The pager does not perform this optimization for pages which are going to be both read and written.)

5. Invokes the copy-on-modify routine if Read I/O (write into memory) set the MODIFY bit in the PTE, and if copy on modify is set

6. Increments the reference count in the PFN database for the page

7. Repeats these steps until all pages are locked
7.4.6.5 Unlocking Pages from Memory

The unlock routine has the following interface definition:

```c
PROCEDURE eUnlock_buffer ( 
    IN host_transfer_list : POINTER eHost_transfer_list; 
);
```

Routine description:

This routine unlocks pages from memory which were locked in memory by the eProbe_and_lock_buffer routine.

Arguments:

IN host_transfer_list - pointer to a host transfer list, the page_frame_number array is set to -1's.

Return value:

none - PFN elements in host transfer list are set to -1.

7.4.6.5.1 Steps to Unlock Pages from Memory

Memory management software uses the following steps to unlock pages from memory:

1. Gets the next page frame number using the host transfer list Decrement the reference count of the page (If it is less than zero, Mica issues a bug check.)
2. Puts the page on the modify, standby or free list if the reference count is zero, depending on the contents of the reference_pte element and the state of the modified bit
3. Sets the page frame number to -1, indicating the page is no longer locked in memory
4. Repeats until all pages are unlocked

7.5 PFN Database

The PFN database contains information about each page of physical memory. The fact that this information must be available while the page itself is in use prevents this information from being stored in the page. Every page of memory is in one of five states:

- Active and valid
- Transition page on the modified or standby list, or read in progress
- Available page on the free list
- Available page on the zeroed list
- Unusable due to hardware errors (these pages are on the bad list)

The PFN database consists of a 32-byte record for each page of physical memory. Unlike the VMS PFN database, the 32 bytes are grouped together in a record structure. This causes fewer translation buffer (TB) misses per PFN database access, and the 32-byte size allows record addresses within the database to be calculated in a straightforward manner: memory management software gets the page frame number, shifts left 5 bits (multiply by 32) and uses that as the offset into the PFN database. The PFN database must be memory resident at all times. The amount of memory occupied by the PFN database for a 1-gigabyte machine is 4 megabytes.
7.5.1 Pillar Record for PFN Database

mm$PFN : RECORD
  UNION CASE *
    WHEN 1 THEN
      flink : integer [-1..];
      WHEN 2 THEN
        event_address : POINTER k$dispatcher;
    END UNION;
  UNION CASE *
    WHEN 1 THEN
      blink : integer [-1..];
      WHEN 2 THEN
        share_count : integer;
    END UNION;
  reference_pte : type_pointer_to_pte;
  original_pte : mm$pte;
  reference_count : integer;
  state : state_type;
  spare : integer [0..255] SIZE(bit,8);
  page_type : page_type_packed_type;
END RECORD;

type_pointer_to_pte : RECORD
  prototype_pte_offset : integer;
  related_pfn : integer;
END RECORD;

mm$physical_address : large_integer; !## [0..2**45 - 1];

state_type : RECORD
  modify : bit;
  delete_pending : bit;
  io_error : bit;
  report_event : bit;
  bad : bit;
  clustered : bit;
  prototype_pte : bit; ! when set the reference_pte field refers to a
  ! prototype PTE
  ! this allows the field to be referenced using
  ! the VA in prototype_ppte_offset rather than
  ! physically
  rsvl : bit;
  page_location : page_location_packed_type;
END RECORD;

page_type_type : (mm$C_page_private,
  mm$C_page_system_paged,
  mm$C_page_system_nonpaged,
  mm$C_page_shared,
  mm$C_page_shared_readonly,
  mm$C_page_shared_execute,
  mm$C_page_shared_write,
  mm$C_page_segment_1_ppte,
  mm$C_page_segment_2_ppte);

page_type_packed_type : [mm$C_page_private..mm$C_page_segment_2_ppte] SIZE(bit,8);
Figure 7–10 shows the layout of each record in the PFN database.

Figure 7–10: PFN Database Layout

<table>
<thead>
<tr>
<th>Page Type</th>
<th>Reserved</th>
<th>State</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Forward Link or Event Address</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Backward Link or Share Count</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Reference PTE Element</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Original PTE Element</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Reference Count</td>
<td></td>
</tr>
</tbody>
</table>

Table 7–1 summarizes the fields in the PFN database record.

<table>
<thead>
<tr>
<th>Table 7–1: Summary of PFN Fields</th>
</tr>
</thead>
<tbody>
<tr>
<td>PFN Element</td>
</tr>
<tr>
<td>-------------</td>
</tr>
<tr>
<td>FLINK</td>
</tr>
<tr>
<td>BLINK</td>
</tr>
<tr>
<td>REFERENCE_PTE</td>
</tr>
<tr>
<td>ORIGINAL_PTE</td>
</tr>
<tr>
<td>STATE</td>
</tr>
<tr>
<td>PAGE_TYPE</td>
</tr>
<tr>
<td>SHARE_COUNT</td>
</tr>
<tr>
<td>REFERENCE_COUNT</td>
</tr>
<tr>
<td>EVENT_ADDRESS</td>
</tr>
</tbody>
</table>

The following sections discuss the PFN fields in greater detail.
7.5.2 PFN Database: FLINK Element

The FLINK element is the forward link for pages that are on the page lists (zeroed, free, standby, modified, or bad). The FLINK element contains the index of the next PFN record in the list.

7.5.3 PFN Database: BLINK Element

This element is similar to the FLINK element. BLINK points to previous page in one of the page lists. Like the FLINK element, it is 32 bits wide.

7.5.4 PFN Database: REFERENCE_PTE Element (Pointer to PTE)

When a physical page in the standby list is assigned another use (that is, the page is removed from the head of the standby list), the pager must be able to find the PTE that maps the page. The REFERENCE_PTE element contains the virtual address of the page table page, which contains the PTE (prototype or actual) and the page frame number of the page table page which contains the PTE (prototype or actual), or a -1 if the REFERENCE_PTE contents are invalid. The physical address of the PTE can be calculated in a straight forward manner: memory management takes the page frame number, and shifts left by 13 and ORs in the low 13 bits of the virtual address.

7.5.5 PFN Database: ORIGINAL_PTE Element

The original contents of the PTE are stored in this element. When a physical page is assigned another use, all links with the PTE must be broken. The PTE, which leaves the transition state, must now have one of the following to indicate where the contents of the page may be obtained the next time it is referenced: the page file and offset where the page is located in a paging file, or the subsection address which indicates where the page is located in a mapped file.

The FRAME field of the ORIGINAL_PTE element contains the system virtual address of a subsection, the page file offset where the page resides in a paging file, or a zero. If the FRAME field is zero, and the page has been modified, it must be written to a paging file. This occurs when the page is removed from memory, and the MODIFY bit is set in the STATE element (see below).

When a page that is not copy on modify is modified for the first time after being read into memory, the MODIFY bit is set in the STATE element and the ORIGINAL_PTE element is examined. If the ORIGINAL_PTE element is in page file offset format, the specified page in the paging file is deallocated and the page file offset is zeroed, indicating the page needs a paging file location assigned. If the page is copy on modify, a new page must be obtained, and the MODIFY bit set for the new page in the STATE element, and the ORIGINAL_PTE element indicates the page is destined for a paging file. (See Section 7.8.4.6 for more information.)

This element is 64 bits wide.

Figure 7–11 shows the possible formats for the ORIGINAL_PTE element.

7.5.6 PFN Database: STATE Element

The PFN STATE element indicates the physical state of each physical page. Possible states include page on zeroed list, page on free list, page on standby list, page on modified list, page on bad list, release of page pending, read error on page read, write in progress, read in progress, page is active and valid. This element is 16 bits wide.

The STATE element format is shown in Figure 7–12.
Figure 7–11: Contents of PFN ORIGINAL_PTE Element

PFO set

<table>
<thead>
<tr>
<th>31</th>
<th>13 12 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Offset &lt;18:0&gt;</td>
<td>Reserved</td>
</tr>
<tr>
<td>E</td>
<td>C</td>
</tr>
<tr>
<td>Cluster Factor</td>
<td>Page File Index &lt;12:0&gt;</td>
</tr>
<tr>
<td>63 62 61 60 59 58 57 56 55 54 53 52 51</td>
<td>45 44</td>
</tr>
</tbody>
</table>

PFO set

<table>
<thead>
<tr>
<th>31</th>
<th>13 12 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Zero</td>
<td>Reserved</td>
</tr>
<tr>
<td>K</td>
<td>E</td>
</tr>
<tr>
<td>Cluster Factor</td>
<td>Zero</td>
</tr>
<tr>
<td>63 62 61 60 59 58 57 56 55 54 53 52 51</td>
<td>45 44</td>
</tr>
</tbody>
</table>

PFO clear

<table>
<thead>
<tr>
<th>31</th>
<th>13 12 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>SVA of Subsection &lt;18:0&gt;</td>
<td>Reserved</td>
</tr>
<tr>
<td>K</td>
<td>E</td>
</tr>
<tr>
<td>Cluster Factor</td>
<td>SVA of Subsection &lt;31:19&gt;</td>
</tr>
<tr>
<td>63 62 61 60 59 58 57 56 55 54 53 52 51</td>
<td>45 44</td>
</tr>
</tbody>
</table>

Figure 7–12: Contents of PFN STATE Element

<table>
<thead>
<tr>
<th>15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reserved</td>
</tr>
</tbody>
</table>

Code

- mm$C_loc_zeroed_list
- mm$C_loc_free_list
- mm$C_loc_standby_list
- mm$C_loc_modified_list
- mm$C_loc_bad_list
- mm$C_loc_release_pending
- mm$C_loc_read_error
- mm$C_loc_write_in_progress
- mm$C_loc_read_in_progress
- mm$C_loc_active_and_valid

Location of Page

- Page is on zeroed page list
- Page is on free page list
- Page is on standby page list
- Page is on modified page list
- Page is on bad page list
- Release pending
  - (when REFERENCE_COUNT is zero put page on standby or modified list)
- Read error occurred while page Read was in progress
  - (exact details of handling read errors are TBD)
- Write in progress by modified page writer
- Read in progress by page fault handler
- Page is active and valid

7–22 Memory Management
MODIFY If set, means the page was modified.
DEL If set, means thePFN database contents for the page should be deleted when the reference count goes to zero, and the page should be put on the free list.
IOE I/O error occurred on page during inpage operation.
BAD Bad page, put on bad page list when REFERENCE_COUNT is zero.
REP Report event on Write I/O completion from page writer.
CLU Page was in-paged as part of a page fault cluster.
PRO PTE referred to in reference PTE field is a prototype PTE.

7.5.7 PFN Database: PAGE_TYPE Element

This element distinguishes the different type of valid pages. Valid types include process page, system page, shared read-only page, shared execute-only, shared read/write page, segment 1 page table page and segment 2 page table page. This element is 8 bits wide.

7.5.8 PFN Database: REFERENCE_COUNT Element

The REFERENCE_COUNT element indicates the number of reasons a page should not be put on the standby, free, or modified list. The REFERENCE_COUNT is incremented when a page initially becomes active and when I/O is being performed to the page. It is decremented when the SHARE_COUNT goes to zero and upon I/O completion. When the REFERENCE_COUNT goes to zero, the page should be placed on the standby, free or modified list. This element is 32 bits wide.

7.5.9 PFN Database: SHARE_COUNT Element

The SHARE_COUNT element keeps a count of the number of sharers for the page. It counts the number of process page table entries that map a particular page. SHARE_COUNT is incremented when the page is added to a working set and decremented when the page is removed from a working set.

SHARE_COUNT is also used to count the number of valid PTEs in segment 1 and segment 2 page table pages. When a segment 2 page table page is initially allocated, the REFERENCE_COUNT is one. The SHARE_COUNT is incremented each time a PTE within that page table page is made valid, and decremented when the PTE is invalid and not in a transition state. If the SHARE_COUNT goes to zero, the segment 2 page table page is eligible for replacement in the working set. The SHARE_COUNT element overlays BLINK, since an active and valid page is not on any lists.

7.5.10 PFN Database: EVENT_ADDRESS Element

The EVENT_ADDRESS field overlays the FLINK field. It contains the address of a dispatcher event object during in paging operations.

7.5.11 Page Lists

When a page is freed from memory for any reason, it is placed on a list based on the validity of the contents of the page and the MODIFY bit in the PFN STATE element. If the page contents are no longer valid, as in the case in which a program calls the Delete Virtual Address service for a nonmodified page, the page is placed on the free list. If the page contents are still valid, the page is placed on one of the transition lists, either the modified or the standby list, depending on the contents of the MODIFY bit in the PFN STATE element.

If a fault occurs for a page on a transition list (either standby or modified page list), the pager unlinks the page from the list and makes it available to the faulting process. Thus, the standby and modify page lists act as a cache for the most recently discarded pages, in addition to being a source of available pages. At the time a page is removed from the head of the standby page list and filled with new information, the PFN database and the page table entries for both the old and the new pages must be updated.
The Pillar definition of a page list is shown below:

```
mm$spfn_list : RECORD
  flink : integer; !first page in the list
  blink : integer; !last page in the list
  count : integer; !number of pages in the list
  list_type : page_location_type;
END RECORD;
```

The page lists are linked by page frame numbers through the PFN database, rather than through data in the page itself. The PFN database is used for linking because the page still has valid contents, even though it is on one of the page lists.

The lists are maintained using the FLINK and the BLINK structures in the PFN database and are discussed in the following sections.

### 7.5.11.1 Zeroed Page List

The zeroed page list is a doubly linked list of physical memory pages that contain all zeros. Pages are added to the tail of the list by the zero page thread. When the zero page thread is activated, it attempts to remove a page from the free list. If a page on the free list is removed, the zero page thread then zeroes the page, and upon completion, adds the page to the tail of the zeroed page list. This continues until there are no pages on the free page list, since the zero page thread has nothing better to do.

Associated with the zeroed page list is a list head that contains the page frame number of the first and last pages on the list, and a count of the number of pages in the list.

### 7.5.11.2 Free Page List

The free page list is a doubly linked list of physical memory pages that are available for use. Pages are added to the tail and always removed from the head. A page is placed on the free page list when its reference count in the PFN database becomes zero, the MODIFY bit in the PFN STATE element is clear, and the REFERENCE_PTE contents are zero.

Associated with the free page list is a list head that contains the page frame number of the first and last pages on the list, and a count of the number of pages in the list.

### 7.5.11.3 Standby Page List

The standby page list is a doubly linked list of physical memory pages that are available for use. Pages are added to the tail and removed from the head, or in the case of a fault of a page on the standby list, removed from an arbitrary place. A page is placed on the standby page list when its reference count in the PFN database becomes zero, the MODIFY bit in the PFN STATE element is clear, and the REFERENCE_PTE element is valid. The clear MODIFY bit indicates the page has not been written into, and the disk retrieval information in the PFN database is still valid.

Associated with the standby page list is a list head that contains the page frame number of the first and last pages on the list, and a count of the number of pages in the list.

7–24 Memory Management
7.5.11.4 Modified Page List

The modified page list is a doubly linked list of physical memory pages that need to have their contents written to their backing store address before reuse. A page is placed in the modified list when its reference count in the PFN database goes to zero, and the MODIFY bit in the PFN STATE element is set. When the MODIFY bit is set, it indicates that the page has been altered since it was last read into memory from an image file, paging file, or mapped file, or the page was demand zero.

A page is removed from the modified page list by the Modified Page Writer and written to its backing store address. Upon completion of the write, if the REFERENCE_COUNT is zero, the page is linked into the standby list if the REFERENCE_PTE contents are valid, or the free list if the REFERENCE_PTE is invalid.

Just as pages can be faulted from the standby page list for reuse, pages can be faulted from the modified page list. If the write has not been started, the page is unlinked from the list. Even if the write is in progress, the page can still be removed from the modified list.

The writing of modified pages causes the reference count for the page to be increased. When the write completes, the reference count is decremented, and if the page has been faulted back into a process's working set, the count is nonzero. Pages with nonzero reference counts are not put on the standby or free list.

Associated with the modified page list is a list head that contains the page frame number of the first and last pages on the list, and a count of the number of pages in the list. Memory management also maintains a high and low limit count for the modified page list. When the number of pages on the modified list exceeds the high limit, the modified page writer is signaled to write some of the modified pages.

7.5.11.5 Bad Page List

The bad page list is similar to all the other lists in structure. It links together all the pages of memory that are considered unusable due to memory parity errors.

7.5.12 Retrieving a Free Page

The pager needs to provide a page of memory to an address space when:

- A demand zero page is required
- A copy-on-modify page is required
- A page is to be read from disk

For the case of demand zero, the pager attempts to allocate a page from the zeroed page list. If this is successful, the page is mapped into the requested address space and the page fault is dismissed. If no pages exist on the zeroed page list, an attempt is made to get a page from the free page list. If the page is obtained from the free page list, it is then zeroed and linked into the requesting address space as before. If no page exists on the free page list, an attempt is made to obtain a page from the standby list. If the page is obtained from the standby list, the PTE referred to in the REFERENCE_PTE field of the PFN database is updated, the page is zeroed, and added to the requesting address space as before. If no pages exist on any available lists, the process enters a resource wait state, awaiting a page on either of the three lists.

User mode ASTs are only deliverable to a thread executing in the memory management subsystem when the thread is waiting on a page to be freed. Otherwise, user mode ASTs are disabled until the page fault is satisfied, even if the page is read from secondary storage.

In the case of copy on modify, the first attempt is to get the page from the free list, next from the zeroed list, and lastly from the standby list.
The case of reading a page from the disk is identical to the case of the demand zero request if the amount of data being read from the disk is less than a page (16 disk blocks). However, if a page must be zeroed, only the portion of the page not modified from the disk read is zeroed. In the case of reading a full page from disk, the search is identical to the copy-on-modify case.

No other thread within the process can see the contents of the new page before the page is overwritten. This is accomplished by first mapping the page into a free PTE in Hyper space with the protection as user no access, then overwriting the page, and finally moving the page into the proper place and flushing the appropriate TB caches.

Whenever a page is removed from the standby list to be reused, the PTE referred to in the REFERENCE_PTE field of the PFN database is updated to indicate the page is no longer in transition and the ORIGINAL_PTE element merged into the PTE. Also, the SHARE_COUNT for the page in which the REFERENCE_PTE refers to is decremented, indicating one less active/transition page for the table page or prototype PTE table page.

7.5.13 Paging Files

Paging files are used by the memory management subsystem to store physical page contents, so that a modified page can be reused. The system can have up to 8,191 paging files. Each file can hold a maximum of 524,288 pages, which is the largest value that can be represented in 19 bits. This represents 8,388,608 disk blocks or 8 gigabytes of disk space. A single process is able to use multiple paging files based upon some yet-to-be-determined algorithm (could be file with largest free space or some metric on disk access time).

An invalid PTE (V=0) that refers to a page in a paging file is in page file offset format and the PFO bit in the PTE is set. The specific paging file is represented in the index portion, PTE<44:32>, and the virtual block number divided by 16 is stored in the offset, PTE<31:13>. A paging file is represented using a structure that contains the total number of pages in the page file, the number of pages free, a pointer to a channel for the file, a pointer to the bit map representing the free/used pages within the file, and other information required to manage the paging file.

The retrieval pointers referenced from the file object are allocated from nonpaged pool.

7.6 Address Space Data Structures

7.6.1 Process Control Block

Each process in the system has a process control block (PCB). The process control block for each process contains, among other items, the working set mutex. This mutex provides synchronization for multiple threads in the same address space.

The PCB has a master PTE. The master PTE is a quadword-aligned structure that has the same format as a prototype PTE. The PFN REFERENCE_PTE contents for the physical page that maps the segment 1 page table page contains the address of the master PTE. If the segment 1 page table page is paged out of memory, its backing store address resides in the master PTE. When the process is brought back into the balance set, the master PTE is used to locate the segment 1 page table page.

7.6.2 Address Space Tables

The address space tables consist of the working set list and other structures relating to the management of a process's address space. The address space tables are contained in the second 4-megabyte area of Hyper space, with a virtual address space range FFF00000 to FFFFFFFF, inclusive.
7.6.2.1 Working Set List

Each page fault (translation not valid fault) requires a page to become resident in memory for the faulting address space. The working set list is used to track the current number of pages an address space has in memory. For each page fault, the required page is added to the working set. When the working set reaches its limit, for each new page added, a page must be discarded. The discarded page is not removed from memory immediately, rather it is placed at the tail of either the modified or the standby page list. By placing the discarded page on a transition list, its contents are still valid, and it can be easily faulted back into the working set. In this scenario, the modify and standby lists act as a cache for each working set.

The working set list contains entries to describe that portion of the process's virtual address space that has been used most recently. All valid process pages have an entry in the working set list. The working set list can contain at most 524,288 entries (512 segment 1 PTEs multiplied by 1024 PTEs per segment 1 PTE) which describes 4 gigabytes. At 4 bytes per entry, this occupies (at most) 2 megabytes of the address space tables. However, a typical working set list occupies a much smaller amount; for example, a 16-megabyte working set consumes 2048 pages, which requires an 8192-byte (one page) working set list.

The working set list consists of three regions, as Figure 7–13 shows: the permanently locked portion of the working set list, the pages that are locked by user request, and the dynamic portion of the working set.

The permanently locked portion of the working set list contains the pages that must be resident in memory if the process is active. These include the following structures:

- The kernel stack(s)
- The process control region
- The vector save page(s) (if needed)
- The segment 1 page table page
- The segment 2 page table page(s), which map the kernel stack, PCR and vector save page(s)
- The address space tables
- The working set list

The PTEs that map the address space tables are always resident when the process is in the balance set, since they reside in the second half of the segment 1 page table.

The pages that reside in the locked by user request portion are all pages that are locked by the user with the executive services Lock Pages in Working Set or Lock Pages in Memory.

The dynamic portion of the working set list is the portion used for page replacement. There is a pointer that contains the address of the start of the working set list, and one that contains the address of the end of the working set list, which is also the end of the dynamic region.

There are also two indexes into the working set list. One represents the working set list entry (WSLE) number of the start of the dynamic region. The other represents the WSLE number from the start of the working set (the first is 0) for the last WSLE that was added to the dynamic region. To calculate the address of the last WSLE added to the dynamic portion of the working set list, multiply the index by 4, and add the result to the address of the start of the working set.

The Pillar record definition for working set list is shown below:
Figure 7-13 shows the format of the working set list.

Figure 7-13: Working Set List

The working set list consists of working set list entries (WSLE). Each WSLE consists of the virtual page number in bits <31:13> and flags in bits <12:0>.

Figure 7-14 shows the format of a working set list entry.

Figure 7-14: Contents of Working Set List Entry

Virtual Page Number | Page Type | P | E | D | F | L | K | W | M | V

7-28 Memory Management
If the VALID bit in the WSLE is clear, and the working set has not reached its authorized size, then the entry can be reused without removing a page from the working set.

When a process virtual address space PTE is valid, there is a valid WSLE in the working set list that has the virtual page number of that PTE. The only exception is for pages which are in the nonpaged portion of system space. Since these pages are never faulted, they are never added or removed from the process's working set. The pager uses the working set list to enforce the rule that a process pages against itself.

When a page fault occurs, and the address space currently occupies all of the physical memory that it is allowed (that is, the working set is full), the pager gets the index that points to the WSLE most recently added to the list. The index is then incremented, and converted to an address. The calculated address is compared with the address for the end of the working set. If the calculated address exceeds the end of the list, the address is reset to the address of the start of the dynamic region, and the index becomes the WSLE number of the start of the dynamic region.

The working set list entry at the calculated address is now checked to see if it is eligible for replacement. This replacement algorithm is a first-in/first-out scheme; however, the current page referenced by the current WSLE might not be the page removed from the working set.

A page in the working set is not considered eligible for replacement if it is locked in memory, locked in the working set, is a segment 2 page with valid PTEs or transition PTEs, or is in the TB cache. Pages in the TB cache are considered recently used pages, and therefore should not be removed from the working set, since there is a high probability they will be faulted back into the working set. A page's status in the TB cache is determined by using the contents of the working set list entry and the MMPR TBCHK instruction.

The working set list search continues until a candidate for replacement is found, or a maximum number of entries are scanned, at which time the first removable entry from where the search started is the candidate. The selected WSLE indicates which page is removed from the working set. The page is removed by invalidating the PTE. If the removed page is shared, that is, refers to a prototype PTE, the SVA of the prototype PTE is copied from the PFN database REFERENCE_PTE element into the PFN field of the PTE. The SHARE_COUNT of the page is decremented, and if it is now zero, the REFERENCE_COUNT is decremented. Since a PTE became invalid and nontransitional, the SHARE_COUNT for the page table page which contains the PTE is also decremented.

If the SHARE_COUNT is zero, the prototype PTE is made invalid by clearing the VALID bit and setting the TRANSITION bit, indicating that the page is in a transition state. If the REFERENCE_COUNT went to zero, the page is then added to the tail of the modified or standby list, depending on the setting of the MODIFY bit in the PFN STATE element.

If the page is a process-private page, then the PTE is made invalid by clearing the VALID bit and setting the TRANSITION bit in the PTE, indicating that the page is in a transition state. The SHARE_COUNT for the page is decremented and must now be zero, because the page is private page.
The REFERENCE_COUNT is decremented, and if it is zero, the page is added to the tail of the modified or standby list, depending on the setting of the MODIFY bit in the PFN STATE element for the page. If the page was modified, the PFN ORIGINAL_PTE element is examined; if it is in page file offset format and the PTE FRAME field is nonzero, the page in the paging file referred to by the ORIGINAL_PTE element is deallocated, and the FRAME field in the ORIGINAL_PTE element is zeroed. Refer to Section 7.12.1 for more information.

Once the page represented by the working set list entry is removed from the working set, the working set entry's VALID bit is cleared and the READIED bit is set. The READIED bit indicates that this WSLE has been freed, but the page which will replace it is not yet valid, and therefore not in the working set.

The contents of the PTE for the new page (which is going to be added to the working set) are examined, and appropriate actions are taken to make the page valid. Once the page is valid, the PTE contains, in the low 14 bits of the software flags (PTE <55:45>), the low 14 bits of the pointer into the working set list for that virtual page. This means that for a given valid virtual address, the WSLE can be found in (working set list size)/16384 lookups. Therefore, for a working set less than 16,384 pages (128 megabytes), the index is direct. This greatly improves the speed of virtual address deletion.

The page replacement for the working set list uses a modified first-in/first-out algorithm. The algorithm is simple, has low overhead, and allows pages that are removed from the working set to be easily faulted back into the working set if the page is still on the standby or modified page list.

### 7.6.3 Address Space Numbers

The PRISM architecture allows for the optional implementation of address space numbers (process tags). On processors that support address space numbers, address space numbers (ASNs) are assigned globally across the system.

The use and management of ASNs is the providence of the kernel, and is described in Chapter 4, The Kernel.

Address space match (ASM) is used for all portions of the executive, as this is the only portion of the address space that is shared in the exact same state in every address space. The setting of ASM in the paged portion of the system space allows threads to reference portions of the page executive that are not in their working set, but are in the translation buffer, without incurring a page fault.

### 7.7 System Services for Mapping Address Space

The following services provide the mechanism to create portions of the address space which may be shared with other processes. The basic unit of sharing is called a section. A section is a disk file, portion of a disk file, or a paging file. A disk file section may be created as a data file or as an image file.

#### 7.7.1 Create Section Service

The Create Section memory service creates a section in the specified object container or the process private container if no object container is specified. The exec$create_section has the following format:

```
PROCEDURE exec$create_section ( 
    OUT section_id : exec$object_id; 
    IN item_list : ITEM LIST CONFORM OPTIONAL; 
    IN file_channel : exec$channel OPTIONAL; 
    IN byte_offset : integer OPTIONAL; 
    IN size : integer OPTIONAL; 
    IN virtual_block_number : integer OPTIONAL; 
    IN protection : exec$protection_code OPTIONAL; 
    IN identification : exec$section_ident OPTIONAL; 
) RETURNS STATUS;
```
If the user wants to create a section which maps a file, the file (which may be an image file or a data file) must already be opened, allowing file access checks to be performed by the file system, and initializing a channel for file access.

Each user may map the same section at a different virtual address, and the same user may create multiple sections for the same file using different ranges of the file, but the mapping between virtual block number and relative page number is always the same.

Users can only map on 64-Kbyte virtual address boundaries. For a data file, this means that the first VBN of a file, VBN 1, can be mapped only at a 64-Kbyte boundary in the address space. An attempt to map VBN 2 will cause VBN 1 to be mapped as well, and aligned on a 64-Kbyte boundary. Thus, the granularity of data file mapping is 64 Kbytes. However, less than 64 Kbytes can be mapped; the remainder are set no access and filled with zeros, as appropriate.

The Create Section service causes a segment object to be created for the section if one does not exist. The segment object consists of a segment header which contains the prototype PTEs, and a control area which contains zero or more subsections.

The segment header is part of the segment object body and resides in paged pool. The segment header contains the segment type (data or image), the number of pages in the segment, the prototype PTEs and a pointer to the control area.

The control area resides in nonpaged pool, and consists of a channel pointer, the number of PFN database references to prototype PTEs within this segment, the number of PTE references to this segment, a pointer to the segment object body, and zero or more subsections.

A control area has zero subsections only if it maps a pagefile section, and has more than one subsection only if it maps an image file.

Each subsection contains the information necessary to locate the virtual blocks which correspond to the prototype PTE.

The segment object is a memory management object which the user cannot create directly; it is built as a result of a Create Section service. The segment object resides in a system level object container.

Figure 7–15 indicates the relationship between structures when a Create Section service is issued.

### 7.7.1.1 Subsection Descriptor

The subsection descriptor in the segment control area contains a pointer to the control area for the mapped file, the base virtual address of the first prototype PTE that points to this subsection, the base virtual block number for this subsection, the number of pages in this subsection, and the number of virtual blocks in this subsection. Each subsection is aligned on a 64-Kbyte boundary. This alignment ensures image compatibility with page sizes up to 64 Kbytes.

When a section is created specifying the section type as image, the image section descriptors (ISDs) in the image header are analyzed to create subsection descriptors.

If no file channel is specified, a page file section is created. When a page file section is deleted, and no other processes are accessing the section, all data within the section is lost. Contrast this to a section which is backed by a file. When backed by a file, the section is deleted and no processes are referencing the section, any pages referring the the section in memory are written back to the file.
7.7.2 Map Section Service

The Map Section executive service allows a thread to map a previously created section into its virtual address space.

The Map Section service also builds a mapping object in the process private object container. This object contains the virtual address range of the mapping, the number of pages in the mapping, and a pointer to the section object that was mapped.
PROCEDURE exec$map_section (  
  OUT mapping_id : exec$object_id;  
  IN item_list : ITEM LIST CONFORM OPTIONAL;  
  IN section_id : exec$object_id;  
  IN desired_address : exec$address_range OPTIONAL;  
  OUT actual_address : exec$address_range OPTIONAL;  
  IN protection : exec$protection_code OPTIONAL;  
  IN identification : exec$section_ident OPTIONAL;  
  IN relative_page : integer OPTIONAL;  
) RETURNS STATUS;

Once a section has been mapped, its virtual address space can only be deleted by deleting the mapping object. An attempt to delete the virtual address space by use of exec$delete_virtual_address_space returns an error.

7.7.2.1 Flow of Create Section

When the Create Section executive service is issued and a channel is specified, the service uses some TBD information within the channel to obtain a name (note this name does not have to be an ASCII name, it could be the address of a file object). The Translate Object Name service is called to see if a segment already exists for the specified channel. If a segment already exists it is referenced using the exec$reference_object_by_id service. If exec$reference_object_by_id succeeds, the section is created using information from the system service arguments and the found segment object.

If exec$translate_object_name or exec$reference_object_by_id fails, the file does not have an associated segment. A segment is created using the channel TBD mechanism to obtain the name.

If the section specifies the file is a data file, the prototype PTEs are built to map the complete file, with each having the same protection and referring to the same subsection. Image file sections, however, may have multiple subsections, each subsection representing a different protection.

The section object defines the portion of a file which may be mapped. In the case of mapping an image file or a paging file, the complete range is mapped. This is done by indicating the range of prototype PTEs within the segment object that the section refers.

When a Map Section service is issued, a mapping object is created which contains a pointer to the section object, and the virtual address ranges of the section. The mapping object is used by the debugger to find where images are mapped in the virtual address space. It is also used by the mapping object delete routine to locate the range of virtual address which should be deleted.

During initialization of the segment 2 PTEs for the Map Section service, each PTE has the VALID bit cleared, and contains the system virtual address of the prototype PTE that maps the page. The protection and some flags are copied from the prototype PTE that maps the page.

Hence, at initial mapping of a section, each PTE within the mapped section contains a system virtual address that is 8 greater than the contents of the previous PTE in the section. The system virtual address of the first PTE in the segment points to the first prototype PTE of the segment if the requested starting offset within the segment is zero or no offset was specified.

7.7.3 Deletion of Mapping Objects, Sections and Segments

A mapping object is deleted explicitly by a call to exec$delete_object_id, specifying the mapping ID or implicitly by the process's private container being deleted. Mapping objects are nontransferable. The mapping object's delete routine dereferences the pointer to the section object, and deletes the virtual address space specified by the mapping object.

A section object is deleted explicitly by a call to exec$delete_object_id, specifying the section ID or implicitly by the object container containing the section object being deleted. Section objects may be transferred. The section object's delete routine is not invoked until all mapping objects which refer to the section have been deleted.

The section object's delete routine releases the pointer to the segment object.
The segment object, which is not visible to user mode, is not deleted until the following are true:

- No sections reference the segment object. This implies that no mapping objects for the segment exist so no process PTEs refer to any prototype PTEs within the segment object.
- No REFERENCE_PTE elements within the PFN database refer to any prototype PTEs within the segment object.

The segment object's delete routine deletes the system channel to the file providing that the segment was not a page file segment, and deallocates the storage for the control area.

7.7.4 Prototype PTE

Prototype PTEs have the same format as regular PTEs, but are a software-only structure; that is, they never reside in a page table page. Prototype PTEs are used to facilitate sharing of pages.

Prototype PTEs are created by the exec$create_section executive service. These prototype PTEs become the master page table of a segment, allowing multiple users to map and share the segment. There is a prototype PTE for each page of a segment, and the prototype PTEs must be virtually contiguous in the system address space. The segment object describes the number of prototype PTEs in the segment and can locate the first prototype PTE.

When a segment is initially created, all prototype PTEs are invalid, and are in one of the following two formats:

- Subsection format — The system virtual address in the PTE FRAME field points to the subsection that describes the file in which the page resides, and contains the necessary information to calculate the disk address and locate the file.
- Demand zero format — The flags in the prototype PTE indicate that the page does not exist in an image or paging file. When a demand zero page is faulted, the pager provides a page of zeros.

Additional flags within the prototype PTE indicate whether the page is writable, copy on modify, shareable, execute-only, demand zero, or read-only.

As prototype PTEs are referenced, they become valid. When they are made valid, the prototype PTE looks exactly like a valid PTE without the working set index field, with the PFN in the FRAME field and the VALID bit set. The PFN database SHARE_COUNT element for the physical page which is mapped by the prototype PTE represents the number of address spaces that are referring to the same physical page. When the SHARE_COUNT of the physical page referred to by a prototype PTE becomes zero, the prototype PTE enters the transition state by clearing the Valid bit and setting the TRANSITION bit and the REFERENCE_COUNT is decremented. If the REFERENCE_COUNT becomes zero, the page is placed on the modified, or standby lists depending on the state of the MODIFY bit.

If the prototype PTE was demand zero and shared, then when the valid page is removed from memory, the prototype PTE is put into page file offset format, referring to the page that now resides in the paging file.

7.7.5 Locking of Valid Prototype PTEs in Memory

When a prototype PTE becomes valid, the SHARE_COUNT for the physical page which maps the prototype PTE must be incremented. This ensures that the prototype PTE page remains in memory, ensuring valid physical address references to the valid prototype PTE. When the prototype PTE becomes invalid and is not in a transition state, that is, no REFERENCE_PTE elements in the PFN database refer to this PTE, the SHARE_COUNT is decremented for the physical page containing the prototype PTE.
7.7.6 Address Space Deletion

The following actions causes address space deletion:

- Deleting processes
- Deleting threads
- Calling the Delete Virtual Address service
- Performing the Delete Object service for a mapping object

When a page of virtual address is deleted, all resources associated with the page must be returned. These include:

- A physical page for valid and transition pages (if page is private)
- A page file virtual block for pages whose backing store address indicates an allocated block (if page is private)
- A working set list entry for a page in the working set
- Quotas associated with the paging file and working set

Also, shared writable pages that have been modified need to have their contents written to the mapped file if the REFERENCE_COUNT for the page goes to zero.

Pages that have I/O in progress cannot be deleted until the I/O completes or is canceled.

Once all reasons for keeping a page valid have been satisfied, the page is deleted. Deletion of a physical page means that the contents of the PFN REFERENCE_PTE element are cleared, destroying all ties between the physical page and the address space. In addition, the page is placed at the tail of the free list, since the REFERENCE_PTE contents are not valid.

A thread executing in user mode may delete virtual address space by deleting a mapping ID, or by calling exec$delete_virtual_address_space. The routine that actually deletes the address space ensures that the page is owned by user mode. The address space deletion causes any pages that are locked in the working set or locked in memory to be unlocked, and potentially deleted.

Unlike VMS, the address space deletion succeeds on pages which have a REFERENCE_COUNT greater than one (i.e. locked in memory for I/O operations). This occurs because there is no synchronization point at which the code for exec$delete_virtual_address_space can wait for the I/O to complete, as another thread could have been the issuer of the I/O.

Since a page could be deleted while a page is brought in to memory, the pager checks the validity of PTEs anytime it releases then reacquires the working set list mutex.

All routines operating in kernel mode must be prepared to receive an access violation ANY time they refer to an address which is owned by user-mode, even if the page was locked in memory.

7.8 The Pager

The pager consists of a set of memory management routines that execute as a result of a translation not valid fault, that is, a page fault, or a fault on write fault. The pager runs in kernel mode in the context of the process that caused the fault. The pager's function is to make the page for which the fault occurred available in physical memory, so that the process can continue execution. When a page fault occurs, the requested page can be in a mapped file, in memory but not in the process's working set (that is, a transition page), demand zero, or in the paging file.

The pager uses fault on write faults to record the MODIFY bit in the PFN STATE element and for copy on modify. When a fault on write fault occurs, memory management locates the address space's PTE, which is valid. The COM field is examined, and if set, the faulting address space gets a private copy of the page. If the SHARE_COUNT for the physical page is one, then this is the only address space which has the page mapped. In this case, the MODIFY bit in the PFN STATE element is
examined. If the MODIFY bit is clear, the page is made private and the prototype PTE is made invalid and put into subsection format using the information in the ORIGINAL_PTE element in the PFN database.

If the SHARE_COUNT for the page is greater than one or the MODIFY bit is set, then a physical page is removed from the free, zeroed, or transition list. The contents of the page are then copied into the newly acquired page, and the user's PTE is changed to refer to the new page. The MODIFY bit in the STATE element for the new page is set, and the ORIGINAL_PTE element for the new page is placed into page file offset format, with the FRAME field zero. Also, the SHARE_COUNT on the page from which the copy was made is decremented.

Note that it is not possible to have a private page which is copy on modify.

A fault on write fault for a page which does not have the COM flag set in its PTE causes the MODIFY bit to be set in the PFN STATE element for the page. At this time the PFN ORIGINAL_PTE element is examined, and if it indicates the previous contents of the page reside in the paging file, the indicated page in the paging file is freed, as the contents in the paging file are no longer valid, since the copy of the page is memory has been modified.

The pager uses paging files to maintain modified pages from the following sources:
- A write-enabled nonshareable image page or file page (copy on modify)
- A write-enabled shareable page that is demand zero

The pager uses the original file to maintain modified pages for the following:
- A write-enabled shareable mapped file page

Pages that are not write-enabled or modified retain their original backing store addresses.

In order to make pages available as needed, the pager maintains and manipulates the following databases:
- PFN database
- Zeroed page list
- Free page list
- Standby page list
- Modified page list
- Working set list database
- Page table entries

The pager must be able to interpret an invalid PTE in order to create a valid PTE from the databases, yet it must save enough information so that when the page is removed from the working set, an invalid PTE can be reconstructed.

### 7.8.1 PTE Formats for Segment 1 PTEs

A PTE in the segment 1 page table page is one of the following types:
- Active and valid—Indicates that a segment 2 page table page exists for this page.
- Zero—Indicates that no segment 2 page table page exist for this segment 1 PTE.
- Transition—Indicates that the segment 2 page table page is in a transition list.
- Page file offset format—Indicates the backing store address of the segment 2 page table page.
7.8.2 PTE Formats for Segment 2 PTEs

A PTE in the segment 2 page table page is one of the following types:

- Active and valid—Indicates that a page exists for the segment 2 PTE.
- Zero—Indicates that no page exists for this segment 2 PTE.
- Demand zero—Indicates that a page needs to be created and filled with zeros.
- Transition—Indicates that the page referred to by the segment 2 PTE is in a transition list.
- Prototype PTE format—Indicates that the page is resolved from a prototype PTE.
- Page file offset format—Indicates that the page resides in a paging file.

7.8.3 PTE Formats for Prototype PTEs

When a page is initially faulted, the page fault handler interprets the system virtual address, retrieves the prototype PTE, and interprets the prototype PTE. The prototype PTE has one of the following formats:

- Active and valid—Indicates that a page already exists for this prototype PTE.
- Demand zero—Indicates that a page needs to be created and filled with zeros.
- Transition—Indicates that the page for the prototype PTE is in a transition list.
- Subsection format—The PTE FRAME field contains the virtual address of the subsection. This indicates that the page resides in the mapped file (image or data file).
- Page file offset format—Indicates that the page resides in a paging file.

7.8.4 Making PTEs Valid

When a translation not valid fault occurs the pager takes the following steps:

- Finds a working set list entry.
- Finds the PTE which needs to be resolved. This PTE could be a segment 1 PTE, the segment 2 PTE, the segment 1 PTE for the prototype PTE, or the segment 2 PTE for the prototype PTE.
- Validates the found PTE, and dismisses the translation not valid fault.

Once the proper PTE has been located, the contents of the PTE are examined to determine the steps to make the PTE valid. The PTE is in one of the following five states:

- Demand zero
- Page file format
- Subsection format (only a prototype PTE can be in subsection format)
- Transition
- Prototype PTE
7.8.4.1 Demand Zero PTE

If the PTE indicates demand zero, then the a page is obtained from the zeroed page list if possible; otherwise, a page is obtained from either the free list or the standby list, and filled with zeros. The contents of the PTE are modified to be valid, with the page frame number of the page of zeroes. The PFN database REFERENCE_PTE element contains the page frame number and virtual address of the PTE, the MODIFY bit is set, and the PFN ORIGINAL_PTE element is built to indicate that the page is to go to a paging file.

7.8.4.2 Subsection Format of PTE

A page that resides in a mapped file (image or data) is in subsection format. The subsection pointer consists of a 32-bit virtual address of the subsection for the page. The subsection contains the necessary information to calculate the virtual block number and length for the in-page operation.

The virtual block number to read within the mapped file is calculated by subtracting the virtual address of the prototype PTE from the base virtual address for the subsection, and multiplying this result by 2 (16 disk blocks per page, 8 bytes per PTE). This yields the virtual block offset into the file.

To resolve a page fault for a prototype PTE in subsection format, a free page is obtained from the free list, zeroed list or standby list. The PFN database element for the page is marked as "read in progress." The I/O request is generated, and the prototype PTE is marked as "in transition." Once the read completes, the prototype PTE is marked as "valid," with the page frame number stored in bits <44:13>. The previous contents of the prototype PTE are stored in the PFN ORIGINAL_PTE element, and the PFN REFERENCE_PTE element contains the page frame number and virtual address of the prototype PTE.

7.8.4.3 Page File Offset Format

The format for a page in the paging file is page file offset format. The page file index (bits <44:32> of the PTE) is an index into a table of paging files. The table provides a pointer to the allocation bit map for the paging file and the address of the system channel for the file. The offset (bits <31:13> of the PTE) is the virtual page number into the file where the contents of the page are stored.

A free page is obtained and removed from the free list, zeroed list, or standby list, and the PFN database element for that page is marked as "read in progress." The I/O request is generated, and the PTE is marked as "in transition." Once the read completes, the PTE is marked as "valid," with the page frame number inserted in bits <44:13>. The previous contents of the PTE are stored in the PFN ORIGINAL_PTE array, and the PFN REFERENCE_PTE element contains the physical frame number and virtual address of the prototype PTE.

A PTE is in page file offset format if the page resides in a paging file.

7.8.4.4 Transition Pages

If the PTE indicates the desired page is in transition, it is removed from the standby or modified list, as appropriate. This is done by removing the page from the appropriate list, updating the PFN database elements, and increasing the REFERENCE_COUNT and SHARE_COUNT counts for the page by one. Next, the PTE is updated to indicate a valid, nontransitional page.
7.8.4.4.1 Page in Transition with Write in Progress

A page that is in transition with write in progress is currently not on any list because the REFERENCE_COUNT is nonzero. It was incremented when an I/O was issued for the page. Page faults for these "transition" pages can occur. Upon a fault the REFERENCE_COUNT and SHARE_COUNT are incremented, and the state of the page is changed to active and valid.

Although the page has not yet been completely backed up, the assumption is made that the Write will complete successfully. Page faults can thus put the page in the active but unmodified state. In the event of a write error, the I/O completion portion of the modified page writer detects that the page is valid, and sets the MODIFY bit in the STATE element for the page.

7.8.4.4.2 Page in Transition with Read in Progress

It is possible for a page fault to occur for a page that is already being read in from disk. Such a page is referred to as a collided page. A wait operation is issued on the event which is pointed to by the EVENT_ADDRESS field of the PFN database element for the page. When the I/O for the page completes, the event is set causing all threads which are awaiting the collided page to become computable. For more information, see Section 7.10.

7.8.4.5 Prototype PTE

If the PTE in question has the PROTOTYPE_PTE bit set, then the page frame number of the PTE contains the virtual address of the prototype PTE for this page. The PTE to resolve is set to the address of the prototype PTE and that PTE is resolved as described above (it may currently be valid).

Once the prototype PTE is valid, the actual PTE is made valid by copying the page frame number from the prototype PTE, setting the working set index field and the VALID bit of the PTE. The SHARE_COUNT for the valid page is incremented as another working set contains the page.

The SHARE_COUNT of the page which maps the user's segment 2 page table is also incremented indicating one more page is now active in the page table. This prevents the page table from migrating out of memory.

Note that in the case of a PTE which refers to a prototype PTE, none of the software flags in bits <45:58> are examined. When the PTE is made invalid, the contents of these bits which was overlayed by the working set index do not need to be restored, only the frame which contains the virtual address of the prototype PTE needs to be restored. Hence, the old PTE contents do not need to be saved, as the contents of the frame field were already saved in the PFN REFERENCE_PTE element when the prototype PTE became valid initially.

7.8.4.6 Valid Page with Copy On Modify (Fault on Write)

All pages initially brought into the process's working set that are nonshared, but writable, are set as copy on modify, and the FOW bit is set in the PTE. This allows the page to be shared in a read-only state. A fault on write fault is taken when the process attempts to write the page, causing the memory management subsystem to be entered. The memory management subsystem takes one of two actions. If SHARE_COUNT is greater than one, it makes a copy of the page, breaking all the ties with the old page. If no one else is using the old page (SHARE_COUNT is one), it makes the old page a private page. This is done by removing the ties to the prototype PTE and converting the prototype PTE into an invalid PTE, using the contents in the ORIGINAL_PTE field of the PFN database.
7.9 Kernel Mode Entry Pages

A kernel mode entry page provides the Change Mode to Kernel mechanism. A kernel mode entry page consists of 2,048 kernel entry vectors, with each kernel entry vector occupying 4 bytes. A call to a system service results in a JSR to a kernel entry vector within a kernel mode entry page. The kernel mode entry pages reside in system space protected to allow user reads and kernel writes. The fault on execute and kernel entry page bits are set within the PTE.

When a JSR instruction with its destination address as a kernel entry entry vector is executed, a fault on execute fault occurs. The fault on execute fault handler, running in kernel mode, examines the PTE, notices the kernel entry page bit is set, and dispatches the system service.

Figure 7–16 shows the format of the kernel entry vector.

Figure 7–16: Kernel Entry Vector

```
31 1 0
```

ADDRESS OF THE SYSTEM SERVICE ENTRY DESCRIPTOR 0 2 8 9

The Chapter 9, System Service Architecture chapter discusses the mechanism for dispatching system services.

This section will be updated to track modifications made to the Chapter 9, System Service Architecture.

7.10 Memory Management Paging I/O

Included in every Thread Control Block (TCB) are two dispatcher objects of type event, and two I/O status blocks for use by the pager. Anytime a page read is issued from a thread, the first free event and IOSB located in the TCB is specified in the request for I/O. Since paging I/O is synchronous, and not AST interruptible, it is not possible for a single thread to issue more than one paging I/O request at a time.

It is, however, possible for a paging I/O operation to incur a page fault. Once the page issues an I/O request a page fault can occur when locating file retrieval pointers. In this case, the second event and IOSB is specified in the I/O request for the second page fault. If the retrieval pointers are not in memory, they are in the paging file, and the paging file's retrieval pointers are always in memory.

The pager indicates the I/O operation is an in-paging I/O by specifying the iso$e_page_read function code to the file system, and specifying the event located in the TCB. The I/O is issued synchronously, performing a wait operation on the specified event. Upon I/O completion the IO_COMPLETION mechanism notices that this is paging I/O, writes the IOSB and sets the event. It does not attempt to deliver a kernel AST.

The setting of the event satisfies the wait thereby allowing the pager to regain control. The pager updates the PFN database, clears the event, updates the appropriate page tables and completes the fault.

This method of paging I/O also improves collided page handling. When a page read is issued for a specified page, that page is marked as in transition, and the page frame number is stored into the PTE. The PFN database for the physical page has its state set to read in progress and stored in the EVENT_ADDRESS element is the address of the event which is set when the page read completes.
Hence, for a collided page fault, the collided thread acquires the PFN database mutex and examines the EVENT_ADDRESS field of the corresponding PFN element. If the field is nonzero, then the page is still being inpaged. The REFERENCE_COUNT for the page is incremented to indicate that another thread is awaiting the I/O completion. A k$release_mutex with wait true kernel call is issued to release the PFN database mutex, and to wait on the event specified in the EVENT_ADDRESS element.

The k$release_mutex with wait true procedure guarantees that no other thread can access the PFN database until the wait on the event has been queued. This prevents a race condition between the release of the PFN database mutex, and the setting of the event by page read completion.

When the page read completes, any of the threads waiting on the read event may be scheduled to run and must be prepared to do the post-page read processing. This implies that the IOSB is at a known offset from the event address so the read status can be examined by whichever thread acquires the PFN database mutex first.

Post page read processing consists of acquiring the working set mutex and the PFN database mutex as well as examining the EVENT_ADDRESS element in the PFN mutex. If the event address is nonzero then this thread is the first to execute after the inpage and must examine the IOSB, update the PTE, clear the EVENT_ADDRESS element, and clear the event. If the IOSB indicates the page read was unsuccessful, the IO_ERROR flag in the PFN database STATE element is set and an in-page error condition is raised.

If the inpaged operation was successful, the address space's page table entry is examined to ensure that it has not changed, and if it still refers to the same page, it is updated, the mutexes released and the translation not valid fault dismissed.

If the EVENT_ADDRESS element is zero the PFN database STATE element for the page is examined and if it indicates a good page, the prototype PTE flag in the PFN STATE element is examined. If the PROTOTYPE_PTE flag is not set, the page fault is dismissed without updating any PTE elements. This is due to the fact that since the page was not in prototype PTE format it was a private page and hence another thread in the address space completed the fault.

If the STATE element PROTOTYPE_PTE flag is set, the prototype PTE is checked to ensure that it is valid. If the prototype PTE is not valid, it is made valid. Note, that the prototype PTE could be in a transition state as the first thread which did post-read processing could have removed the page from its working set, however, the REFERENCE_COUNT prevents the page from being put on any of the page lists.

The address space's page table entry is examined to ensure that it has not been deleted, and if it still refers to the same page, it is updated, the mutexes are released and the translation not valid fault is dismissed.

If the STATE element indicates an inpaged error occurred, the in-page error condition is raised and the page is not made valid.

Note that incrementing the REFERENCE_COUNT for the page prevents the page from being removed from memory before a thread which is awaiting that page can place that page into its working set. When the event is set each thread waiting on the event decrements the reference count once the post-page read processing has been completed.

7.11 Synchronization Techniques

Memory management for PRISM is designed to work in a multiprocessor environment. The nature of multiprocessing requires that no synchronization is performed by raising IPL, which only affects the current processor. Instead, mutexes for various databases are created as required, and synchronization is maintained through the use of these mutexes.
7.11.1 Synchronization Techniques for Working Set Lists

When a translation not valid fault is taken, an attempt is made to acquire the address space’s working set list mutex. If this attempt is successful, memory management continues to make the page valid. If the attempt is unsuccessful, then another thread is executing using the same working set list and owns the working set list mutex. The requesting thread is put into a wait state, awaiting the release of the working set list mutex. The working set list mutex prevents two threads in the same working set (address space) from accessing the working set list at the same time.

Once the working set mutex is acquired, no valid PTEs can become invalid nor can any invalid PTEs become valid within that address space. It is possible, however, for transition pages to become nontransition pages. This could occur when a page on the standby list is removed for another use.

7.11.2 Synchronization Techniques for PFN Database

Any time changes are being made to a list in the PFN database, the PFN mutex must be acquired. Also, when the PFN database is being modified for an active page, the mutex must be acquired. This prevents two or more threads from attempting to allocate the same page from the free page list and other problems.

7.11.3 Synchronization Techniques for Paging Files

Any time a paging file bitmap is examined to remove or add a page, a mutex is acquired for that paging file. The mutex prevents two or more threads from attempting to allocate or deallocate the same pages within the paging file.

7.11.4 Translation Buffer Synchronization

Regardless of whether ASNs are implemented or not, the translation buffer (TB) must be invalidated every time a valid PTE is changed. This change includes making the PTE invalid or changing the protection. When a valid PTE is changed, the TB entry on the current processor is immediately invalidated, using the procedure k$translation_flush_single.

If the address space is executing on multiple processors, an invalidation must be done on all processors before the operation can be completed. If, however, the address space is only on one processor, the TB invalidation on other processors does not need to be done. The k$translation_flush_single procedure handles these cases.

If the page has address space match set, the allprocessors parameter in the k$translation_flush_single is set to true. This causes the translation buffer on all processors to be flushed.

7.11.5 Instruction Stream Synchronization

If an address space enables fault on execute, or changes the protection on a page that is executable to "no read," the instruction stream cache must be flushed. Additionally, if that address space is currently executing on other processors, their instruction stream cache must also be flushed.

The exec$flush_instruction_buffer service allows software, such as the debugger, that writes into the instruction stream to flush the instruction buffer for itself and for other threads in the same address space executing on other processors.
7.11.6 Page Faults at Elevated IPL

No translation not valid faults or FOW faults are allowed at IPL levels greater than 1. At IPL levels greater than 1, mutexes such as the mutex on the PFN database may not be acquired or released, hence there is no way to synchronize access to the PFN database for retrieving a page or setting the MODIFY bit.

These restrictions require that procedures executing at high IPL must execute exclusively out of nonpaged system space and reference data that only resides in nonpaged system space. Data that is locked in the working set, in paged system space or user space, may only be referenced from the address space context in which the data pages are locked.

Note, fault on write faults, even on pages locked in the working set, cause a system bug check if generated at an IPL greater than 1.

7.12 Managing Physical Pages

The PRISM operating system does not allow the amount of physical memory to limit the number of processes allowed in the system. Physical memory is extended by keeping only a subset of the total amount of physical memory required for each process in memory at a given time. The modified page writer and the balance set manager are responsible for ensuring that an ample number of free pages exist on the free page list so the physical memory stays fluid.

The managing of physical pages consists of:

- Writing modified pages which are on the modified page list until the number of pages on the modified page list falls below some threshold
- Obtaining free pages when physical memory is overcommitted
- Maintaining the balance set

7.12.1 Modified Page Writer

The modified page writer is a system thread whose sole responsibility is to write pages that are on the modified page list to their specified backing store. To optimize the I/O, the modified page writer attempts to write as many pages as possible in a single I/O request. As I/O requests complete, the modified page writer moves the written pages from the modified list to the tail of the standby or free list providing the reference count for the page is zero.

When a page on the modified page list is being removed from memory, the PFN ORIGINAL_PTE element contents are examined to determine where the page should be written. If the ORIGINAL_PTE element contains a PTE in page file offset format, the page should be written to a paging file. At this point, the modified page writer assigns a paging file and location within the paging file, and updates the ORIGINAL_PTE element in the PFN database for the physical page.

If the ORIGINAL_PTE element contents are in subsection format, then the page belongs to a mapped file and should be written into the file. The address of the subsection is contained in the ORIGINAL_PTE element and the virtual block number into the file is calculated by using the SVA of the prototype PTE found in the REFERENCE_PTE element. The subsection contains the information needed to do the I/O to the file.

The modified page writer attempts to write as many pages as possible, up to some limit, into a single I/O. The I/O is issued and the modified page writer awaits notification when the write completes.

When the modified page I/O completes, the reference count for the pages written is decreased. If the reference count is zero, the page is placed at the tail of the standby list.
7.12.2 Balance Set Management

The balance set manager is a system thread. As such, it can be selected for execution, just like any other thread on the system. The balance set manager executes entirely in kernel mode and becomes computable by the setting of an event by the memory management subsystem when the total number of free pages, zeroed pages, and transition pages falls below some threshold.

The balance set manager's duty is to increase the number of free pages in memory. This is accomplished by shrinking the number of pages in an address space or removing an address space from the system. The set of all address space working sets currently resident in physical memory is called the balance set.

Since the working set list is maintained in the user's virtual address space, the actual shrinking of the working set must be performed in the context of the process. This is accomplished by issuing a kernel mode AST to a thread in the target process which removes the specified number of pages from the process's working set.

In the event that the amount of memory on the system is severely over-committed, the balance set manager removes an address space from the system, releasing all its active pages, thereby freeing more pages than by trimming the working set. This is accomplished by first preventing all threads in the process from becoming executable.

The balance set manager then maps the segment 1 PTE of the address space into a reserved PTE in its segment 1 page table. The mapping into the balance set manager's process of the PTBR of the address space to remove allows the balance set manager to reference the address space's page tables and working set lists without being in the context of that address space.

The balance set manager examines the address space's working set list, removing pages that are valid, even if the page is locked in the working set. For pages that are not locked in the working set and not locked in memory, the WSLE is cleared. However, for pages that are locked in the working set, the only modification made to the working set list entry is that the VALID bit is cleared, indicating that the page is no longer in the working set. Care is taken during this operation not to release any address space tables, such as the working set list, until all other pages have been released. Note that the segment 1 page table page is not in the working set.

For pages that are locked in memory, the working set list element is not changed, and the page is not removed from the segment 2 page table. In this case, the segment 1 and segment 2 page tables that reference the locked page will not be removed from memory. It does, however, require enhanced identifiers (PSWAPM on a VAX/VMS system) in order to lock a page in memory.

Once all process pages have been removed from the working set, segment 2 page table pages are automatically removed from memory when their SHARE_COUNT goes to zero. It is important to note at this point that the only segment 2 PTEs that have a nonzero SHARE_COUNT are those that contain process-private pages in transition states, or pages that are locked in memory.

The segment 1 page table page is eligible for removal when the only valid PTEs are those that map the segment 1 page table page and the nonpaged system space. Hence, for the segment 1 page table page, SHARE_COUNT is biased by the permanent number of valid pages. When the segment 1 page table page's SHARE_COUNT and REFERENCE_COUNT are zero, the master PTE contains a transition PTE. The REFERENCE_PTE element of the PFN database for the physical page that maps the segment 1 page table page contains the SVA of the master PTE located in the address space's process control block. If and when the segment 1 page table page is removed from memory, the location of the segment 1 page table page in the paging file is stored in the master PTE in the process control block.

Once all pages have been freed from the working set, the balance set manager unmaps the process's PTBR and invalidates the TB entry. All the pages within the address space which we are removing from memory eventually migrates to the modified and free lists.
To restart a process that has been removed from memory, the balance set manager must bring the segment 1 page table page back into memory, if it was removed and remap it into the reserved segment 1 PTE in the balance set manager's address space. The segment 1 page table page of the address space then needs to have the new PTBR contents inserted and made valid, have the master PTE updated, and the PTE for Hyper space restored.

The address space tables are then faulted into memory and the working set list is analyzed. All pages that were locked into the working set are faulted back into memory.

When the locked portion of the working set list is memory resident, the address space is placed into the balance set.

7.12.3 Page Fault Clustering

Page fault clustering is the act of reading more than one page from the disk to satisfy a page fault. For example, if a user program is sequentially accessing a large data array from say, a mapped file, a page fault occurs every time 8 Kbytes of data has been processed. If instead of reading a single page from the disk, a "cluster" of pages is read, a page fault for a data page would occur less frequently.

This technique is successfully used on VMS to overcome the small page size. On PRISM the page size is sixteen times as large as a VAX, but page fault clustering can still offer a substantial performance improvement to those applications which sequentially access code or data.

Let us assume that a typical disk transfer requires 20 milliseconds access time and 4 milliseconds to transfer 8 Kbytes of data. If, instead we transfer 32 Kbytes of data, and assuming the blocks are contiguous, the total transfer time increases by 12 milliseconds, yet we have brought the faulted page into memory and three other virtually contiguous pages. If the user is accessing the pages sequentially, the three extra pages are accessed with no disk I/O, resulting in the elimination of three I/Os averaging 24 milliseconds each.

Unlike VMS, only the page which satisfies the page fault is added to the process's working set. The other pages in the cluster are added to the end of the standby list and the CLU flag is set in the PPN state element.

Placing the additional pages on the standby list and setting the CLU flag has the following advantages:

- At most one page is removed from the working set during a page fault. This results in fewer TB checks and invalidates.
- Pages which are in the working set are not needlessly removed from the working set only to be faulted.
- A method can be developed for heuristically adjusting the cluster factor. For example, if a page with the CLU flag set is removed from the standby list to use as a free page, perhaps the cluster factor for that page should be adjusted or that page should never be clustered. On the other hand, a page which is faulted from the standby list with the CLU flag set could have its cluster factor increased.
7.13 Zero Page Thread

The zero page thread is a system thread responsible for maintaining a generous number of zeroed pages.

At system initialization, the zero page thread is initialized and activated. It runs at priority zero, in kernel mode, and is scheduled when there are no higher priority threads to run and there are pages on the free page list.

When the zero page thread executes, it removes a page from the free list. If no page exists on the free list, the thread goes into a wait state, awaiting a page on the free page list. When a page is obtained from the free page list, the page is mapped and zeroed.

When the page has been zeroed, it is unmapped, the TB entry is invalidated, and the zeroed page is inserted at the tail of the zeroed page list. The zero page thread then attempts to remove another page from the free page list.

This sequence of removing free pages, zeroing the page, and inserting the zeroed page into the zeroed page list is all the zero page thread does.

7.13.1 Performance and Accounting Information Collected

7.13.1.1 System-wide Information

Memory management software collects the following types of performance and accounting information on a system-wide basis.

- Total page faults
- Total page faults for pages owned by kernel mode
- Total page faults for pages owned by user mode
- Total demand zero page faults
- Total number of faults from modified list
- Total number of faults from standby list
- Total number of prototype PTE valid faults
- Total copy-on-modify faults
- Total number of page I/O requests issued
- Total number of pages read
- Total number of write I/O requests issued
- Total number of pages written
- Current number of threads waiting for a page read
- Current number of threads waiting for a collided page
- Current number of threads waiting for a free page
- Size of free list
- Size of standby list
- Size of modified list
- Size of zeroed list
- Size of bad list

7–46 Memory Management
7.13.1.2 **Per Process Information**

Memory management software collects the following types of performance and accounting information on a per process basis:

- Total page faults
- Total page faults for pages owned by kernel mode
- Total page faults for pages owned by user mode
- Total demand zero page faults
- Total number of faults from modified list
- Total number of faults from standby list
- Total number of prototype PTE valid faults
- Total copy-on-modify faults
- Total number of faults from the page file
- Total number of faults from a mapped file
- Total number of fault on writes
- Total number of copy on modifies
- Total number of copy-on-modify actual copies

The cost of collecting these statistics is not free, in fact on VMS, it has been demonstrated that each memory reference reduces the time for a transition fault by 1/2 percent.

7.14 **Allocation of Pool from Paged and Nonpaged Pool**

The procedure `e$allocate_pool` has the following interface:

```plaintext
PROCEDURE e$allocate_pool (  
    IN pool_type: e$pool_index;  
    IN number_of_bytes: integer;  
    IN wait_for_free: boolean;  
    OUT starting_block_address: POINTER anytype;  
) RETURNS STATUS;
```

The `e$allocate_pool` procedure allocates paged or nonpaged memory. If the request is equal to or larger than a page, it is satisfied with a page aligned block of pool.

When a pool request requiring one or more pages is issued, an internal memory management system routine is called. This routine acquires a mutex and checks a bit map to find the requested number of pages. It then declares those pages as in use and releases the mutex. The page table entries (prototype page table entries in the case of paged pool) are modified for the first and last pages in the allocation. For the first page in the allocation, its PTE has the START_OF_ALLOCATION bit set, while the last page has its END_OF_ALLOCATION bit set. These bits are used to deallocate the pool.

The pool deallocation routine has the following interface definition:

```plaintext
PROCEDURE e$deallocate_pool (  
    IN starting_block_address: POINTER anytype;  
); EXTERNAL;
```

When using this routine the STARTING_BLOCK_ADDRESS parameter must be the same value that was returned by `e$allocate_pool`.

The pool allocation and deallocation routines must be called at IPL 0 or IPL 1. Calling the routines above IPL 1 will corrupt the pool database.
7.15 Memory Management Executive Routines for I/O support

Memory management provides routines for locking and unlocking pages in memory, probing virtual address ranges for read or write, and determining if a given virtual address is in nonpaged pool. These routines are:

```plaintext
PROCEDURE e$probe_and_lock_buffer (  
    IN host_transfer_list : POINTER e$host_transfer_list;  
    IN access_mode : mm$mode_type;  !k$k_user or k$k_kernel  
    IN operation_type : e$io_operation;  ! (e$io_read_into_buffer, e$io_write_from_buffer  
                ! e$io_modify_buffer)
);  
PROCEDURE e$unlock_buffer (  
    IN host_transfer_list : POINTER e$host_transfer_list;
);  
PROCEDURE e$get_number_of_pages_for_hlt (  
    IN starting_virtual_address : POINTER anytype;  
    IN length_in_bytes : integer [0..];  
) RETURNS number_of_pages : integer;
PROCEDURE e$is_address_in_nonpaged_pool (  
    IN virtual_address : POINTER anytype;  
) RETURNS boolean;
PROCEDURE e$probe_access_read (  
    IN starting_virtual_address : POINTER anytype;  
    IN length_in_bytes : integer [0..];  
    IN access_mode : mm$mode_type;  !k$k_user or k$k_kernel  
) RETURNS boolean;
PROCEDURE e$probe_access_write (  
    IN starting_virtual_address : POINTER anytype;  
    IN length_in_bytes : integer [0..];  
    IN access_mode : mm$mode_type;  !k$k_user or k$k_kernel  
) RETURNS boolean;
```

7.16 Memory Management Support for Page Fault Monitoring

Page fault monitoring is implemented using the object architecture. There is an object of type Page Fault Monitor which has a set of object service routines. There can be at most one page fault monitor object per process. If a process has a page fault monitor object, the process control block contains the address of the object. Also, the page fault monitor object contains the address of the process control block for which it collects statistics.

The following service enables page fault monitoring for a process.

```plaintext
PROCEDURE exec$create_pagefault_monitor (  
    OUT object_id : exec$object_id;  
    IN item_list : ITEM LIST CONFORM OPTIONAL;  
    IN buffer1 : DESCRIPTOR;  
    IN buffer2 : DESCRIPTOR;  
    IN event_id : exec$object_id OPTIONAL;  
    IN ast_procedure : PROCEDURE OPTIONAL;  
    IN ast_parameter_1 : LONGWORD CONFORM OPTIONAL;  
    IN ast_parameter_2 : LONGWORD CONFORM OPTIONAL;  
) RETURNS STATUS;
```
The exec$create_page_fault_monitor object service routine does the following:

1. Creates an object of type page fault monitor in the specified container, typically process private (An error is returned if the process currently has an object type of page fault monitor.)

2. Permanently locks the buffer region used to collect the data in the process's working set (These pages are locked until the object is deleted. Note the WSLE for these locked pages has the PFM bit set preventing the user from deleting the virtual address space containing the buffers. The PFM bit is cleared and the pages are unlocked when page fault monitoring is terminated.)

3. Initializes data structures in the object

4. Gets the address of the object stored into the process control block for use by the pager

The buffers are quadword aligned structures. For efficiency two equal sized buffers should be provided and the buffers should be multiples of 64 Kbytes and page aligned, but this is not required.

7.16.1 Contents of Buffer

The first four longwords in a valid buffer are shown in Figure 7-17.

Figure 7-17: Valid Buffer

<table>
<thead>
<tr>
<th>BUFFER STATUS FLAG</th>
</tr>
</thead>
<tbody>
<tr>
<td>NUMBER OF LOST PAGE FAULTS DUE TO NO BUFFER</td>
</tr>
<tr>
<td>NUMBER OF PAGE FAULT RECORDS IN BUFFER</td>
</tr>
<tr>
<td>FILLER</td>
</tr>
</tbody>
</table>

The BUFFER STATUS flag is used by the user to indicate when a buffer is available for the page fault monitor to write. When the page fault monitor requires a buffer (at either the initial call or upon filling a buffer), it examines both buffers for a buffer status flag of zero. If it finds a buffer with a buffer status flag of zero, it begins filling that buffer with data. When the buffer is filled or page fault monitoring terminated, the buffer status flag is set to a 1, the count of the number of lost pages and the count of pages fault records is written to the buffer, and the user is notified that a buffer is full.

If the buffer was filled normally, i.e. not due to page fault termination, the user is notified by either an AST or an EVENT or both. The user can determine which buffer was returned by examining the first longword of each buffer for a one.

For each recorded page fault three quadwords are stored in the buffer. The first quadword is the object ID of the thread causing the fault, the second quadword is the accumulated CPU time for the thread causing the fault, and the last quadword consists of the faulting PC, the related virtual address and the type of fault. The last longword containing the related virtual address and the type of fault consists of bits <31:7> of the virtual address and 7 bits of faulting type.

The type of fault is stored in the low order 7 bits of the related VA in page longword.
Figure 7-18: Page Fault Information

<table>
<thead>
<tr>
<th>OBJECT ID OF FAULTING THREAD</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACCUMULATED CPU TIME FOR FAULTING THREAD</td>
</tr>
<tr>
<td>PC OF FAULTING INSTRUCTION</td>
</tr>
<tr>
<td>RELATED VIRTUAL ADDRESS IN PAGE</td>
</tr>
</tbody>
</table>

Bit | Type of Fault (bit set) |
--- |------------------------|
0   | STORE instruction issued, i.e. write to memory |
1   | Demand zero |
2   | Transition fault from standby list |
3   | Transition fault from modified list |
4   | Collided page |
5   | Kernel entry point |
6   | Copy-on-modify fault |

The AST_PROCEDURE parameter is optional. If provided, the specified procedure is invoked when a buffer is full. If an AST_PROCEDURE is not specified, the user waits on the event specified in the exec$create_page_fault_monitor call. This would typically be done in another thread.

Page fault monitoring is terminated by deleting the page fault monitor object. The page fault monitoring type specific delete routine unlocks the pages from the working set, clears the pointer in the process control block and clears the page fault monitor bit in the WSLE for the buffers. The buffer currently being filled is updated with the current counts giving the user the partially filled buffer. If no buffer is currently being filled, the lost data count will not be returned to the user.

7.16.2 Example of Using Page Fault Monitoring

This example assumes a thread has started another thread to capture the page fault monitoring data. This code segment represents the thread capturing the data.

```plaintext
<x>(Page fault monitoring)

VARIABLE
buffer1 : long_data(16384);
buffer2 : long_data(16384);
buffer_pointer : pointer longword conform;
id : exec$object_id;
event : exec$object_id;
lost : integer;

BEGIN

! Create event and other initialization...
!
! turn on page fault monitor for process
!
buffer1[1] = 0;
buffer2[1] = 0;
exec$create_pagefault_monitor (object_id = id,
buffer1 = buffer1,
buffer2 = buffer2,
event_id = event);

LOOP

7-50 Memory Management
7.17 Memory Management Executive Services

The following executive services are provided by memory management routines or directly affect the behavior of memory management.

7.17.1 Adjust Working Set Limit

PROCEDURE exec$adjust_working_set_limit (  
  IN byte_count : integer;  
  OUT working_set_limit : integer OPTIONAL;  
) RETURNS STATUS;

The Adjust Working Set Limit service adjusts a process's current working set limit by the specified number of bytes and returns the new value to the caller. A negative value for the byte count will reduce the maximum size of the working set.

7.17.2 Create Virtual Address Space

PROCEDURE exec$create_virtual_address_space (  
  IN desired_address : exec$address_range;  
  OUT actual_address : exec$address_range;  
) RETURNS STATUS

The Create Virtual Address Space service adds a range of demand zero pages to a process's virtual address space.
7.17.3 Delete Virtual Address Space

PROCEDURE exec$delete_virtual_address_space (  
    IN desired_address : exec$address_range;  
    OUT actual_address : exec$address_range;  
) RETURNS STATUS;

The Delete Virtual Address Space service deletes a range of addresses from a process's virtual address space. Upon successful completion of the service, the deleted pages are inaccessible, and references to them cause access violations.

Only pages which are created as a result of an exec$create_virtual_address_space service may be deleted. If the page was created by mapping a section an error is returned and the page is not deleted.

7.17.4 Expand Stack

The exec$expand_stack service attempts to add the number of bytes specified to the current thread's stack. If the stack cannot be expanded to the limit specified, an error status is returned and the stack is increased as much as possible. The stack limits are reflected in the thread control block.

A user is unable to contract a thread's stack. An attempt to delete virtual address space within the stack returns an error.

PROCEDURE exec$expand_stack (  
    IN additional_bytes : integer [1..];  
    OUT stack_size : integer [1..] OPTIONAL;  
) RETURNS STATUS;

7.17.5 Lock Pages In Memory

PROCEDURE exec$lock_pages_memory (  
    IN desired_address : exec$address_range;  
    OUT actual_address : exec$address_range;  
) RETURNS STATUS;

The Lock Pages In Memory service locks a page or range of pages in memory. The specified virtual pages are forced into the working set, then locked in memory. A locked page is not removed from memory if its process's working set is removed.

7.17.6 Unlock Pages from Memory

PROCEDURE exec$unlock_pages_memory (  
    IN desired_address : exec$address_range;  
    OUT actual_address : exec$address_range;  
) RETURNS STATUS;

The Unlock Pages From Memory service unlocks a page or range of pages from being memory resident. The pages are unlocked from the working set and are eligible for replacement.
7.17.7  Lock Page in Working Set

PROCEDURE exec$lock_page_working_set (  
    IN desired_address : exec$address_range;  
    OUT actual_address : exec$address_range;  
) RETURNS STATUS;

The Lock Page In Working Set service locks a page or range of pages in the working set. If the pages are not currently in the working set, they are brought into the working set and locked. A page that is locked in the working set is not a candidate for replacement.

7.17.8  Unlock Page from Working Set

PROCEDURE exec$unlock_page_working_set (  
    IN desired_address : exec$address_range;  
    OUT actual_address : exec$address_range OPTIONAL;  
) RETURNS STATUS;

The Unlock Page From Working Set service unlocks a page or range of pages from the working set. These pages are now candidates for replacement.

7.17.9  Set Protection on Pages

PROCEDURE exec$set_protection_on_pages (  
    IN desired_address : exec$address_range;  
    OUT actual_address : exec$address_range;  
    IN page_protection : exec$protection_code;  
    OUT last_page_protection : exec$protection_code;  
) RETURNS STATUS;

The Set Protection On Pages service allows a process to change the protection on a page or range of pages.

7.17.10  Lock Process in Balance Set

PROCEDURE exec$lock_in_balance_set (  
) RETURNS STATUS;

The current address space is locked in the balance set and is not eligible for removal by the balance set manager.

7.17.11  Unlock Process from Balance Set

PROCEDURE exec$unlock_from_balance_set (  
) RETURNS STATUS;

The current address space is unlocked from the balance set and is eligible to be removed from the balance set.
7.17.12 Create Section

PROCEDURE exec$create_section (  
  OUT section_id : exec$object_id;  
  IN item_list : ITEM LIST CONFORM OPTIONAL;  
  IN file_channel : exec$channel OPTIONAL;  
  IN byte_offset : integer OPTIONAL;  
  IN virtual_block_number : integer OPTIONAL;  
  IN protection : exec$protection_code OPTIONAL;  
  IN ident : exec$section_ident OPTIONAL;  
) RETURNS STATUS;

7.17.13 Map Section

PROCEDURE exec$map_section (  
  OUT mapping_id : exec$object_id;  
  IN item_list : ITEM LIST CONFORM OPTIONAL;  
  IN section_id : exec$object_id;  
  IN desired_address : exec$saddress_range OPTIONAL;  
  IN protection : exec$protection_code OPTIONAL;  
  IN ident : exec$section_ident OPTIONAL;  
  IN relative_page : integer OPTIONAL;  
  OUT actual_address : exec$saddress_range;  
) RETURNS STATUS;

The Map File service allows a thread to map a portion of its address space with either a specified section of a file, a section of paging file, or an image file.

By use of the MAPPING_TYPE parameter, the caller indicates how the file is to be mapped (as data or image) and the characteristics of the mapping.

7.17.14 Update Mapped Section

PROCEDURE exec$update_mapped_section (  
  IN mapping_id : exec$object_id;  
  IN desired_address : exec$saddress_range OPTIONAL;  
  OUT actual_address : exec$saddress_range;  
  IN flags : exec$update_flags OPTIONAL;  
  IN event_id : exec$object_id OPTIONAL;  
  IN ast_procedure : exec$ast_procedure OPTIONAL;  
  IN ast_parameter_1 : LONGWORD CONFORM OPTIONAL;  
  OUT io_status_block : exec$io_status OPTIONAL;  
) RETURNS STATUS;

The Update Mapped Section service causes pages of a mapped section which have been modified to be written back to the mapped file. If no EVENT_ID, or AST_ADDRESS parameter is specified the service completes synchronously.
7.17.15 Create Page Fault Monitor

The Create Page Fault Monitor executive service provides the user with a mechanism to record the PC, virtual address, and cumulative CPU time for each page fault. It is anticipated that this service would be used by the PCA products for Prism.

```
PROCEDURE exec$create_pagefault_monitor (  
OUT object_id : exec$object_id;  
IN item_list : ITEM LIST CONFORM OPTIONAL;  
IN bufferl1 : DESCRIPTOR;  
IN buffer2 : DESCRIPTOR;  
IN event_id : exec$object_id OPTIONAL;  
IN ast_procedure : PROCEDURE OPTIONAL;  
IN ast_parameter_l : LONGWORD CONFORM OPTIONAL;  
) RETURNS STATUS;
```

7.18 64-Bit Architecture

At some point in the future, the proposed 32-bit PRISM architecture will run out of virtual address bits. When this event occurs, it is likely that the PRISM architecture will be amended to allow for a larger virtual address. With this in mind, portions of the memory management subsystem attempt to anticipate the larger virtual address and minimize the number of changes required within the memory management subsystem.

The current ideas about a larger virtual address space have the notion of a variable page size, which can be 8, 16, 32, or 64 Kbytes per page. A given page size defines the number of bits in the virtual address space. For a page size of 8 Kbytes, the virtual address is comprised of 33 bits; for 16-Kbyte pages, 47 bits; for 32-Kbyte pages, 51 bits; and for 64-Kbyte pages, 55 bits. The format of the PTE will remain constant, but another segment table will be added.

The following data structures and concepts are usable without modification in the larger page size systems:

- Sections are aligned on 64-Kbyte boundaries.
- The PFN database REFERENCE_PTE element is 64 bits, but may need its format modified.

The following data structures and concepts rely on 32-bit virtual addresses and must be changed in the larger virtual address systems:

1. The PTE contains the SVA of the prototype PTE.
2. The prototype PTE contains the SVA of the section.
3. The PFN database FLINK and BLINK contain system virtual addresses.
4. Working set list elements store the virtual page number in 19 bits.
5. Object blocks have system virtual addresses.
6. Kernel mode entry pages contain SVA of the routine.

The resolution of these issues is quite simple. For example, in the case of object blocks, the virtual address field will be increased to 64 bits. In the case of the PFN database and the PTEs, the addresses will not be system virtual addresses, but rather, offsets which are added to a fixed 64-bit base in order to produce a 64-bit virtual address.

The working set list element will be extended to 64 bits. This allows the largest virtual page number, 39 bits, to be stored in the proper place within the quadword, to be used easily as an address.
CHAPTER 8
I/O ARCHITECTURE

8.1 Introduction

This chapter describes the I/O architecture for the Mica operating system. The chapter is divided into three major sections: Overview, Functional Description, and Internal Design. The Overview section provides a brief summary of the I/O system. The Functional Description section provides an operation-level understanding of the various components used by the I/O system. The Internal Design section describes the I/O system from an implementation standpoint.

The following chapters are prerequisite reading for this chapter:

- Chapter 5, Object Architecture
- Chapter 4, The Kernel
- Chapter 6, Process Structure

A note on nomenclature: Since this architecture will be compared to the VMS I/O architecture, an attempt is made to reduce confusion. Mica components with a function identical to a VMS component are named the same, even though the exact implementation might vary. Conversely, for those components that have no VMS counterpart, an attempt has been made to select a name that does not exist anywhere in VMS.

8.2 Overview

The Mica I/O architecture defines the fundamental components of the I/O system, the interface of each component, and the relationships between the components. The objective of this architecture is to provide a framework in which simple or complex I/O structures can be built in an efficient and modular fashion.

The Mica I/O architecture is designed to allow I/O abstractions to be built in successive virtual layers on top of physical or pseudo devices. Examples of these I/O abstractions are file systems, shadowing, striping, and so on.

8.2.1 Function Processors

I/O abstractions and devices are represented by components called function processors. A function processor is an image that contains the code necessary to implement an I/O abstraction (see section Section 8.4.10.1). The purpose of the function processor is to satisfy I/O requests. If an I/O request cannot be completely satisfied by a function processor, then that function processor may pass the request on to a lower-level function processor for further processing.

The function processor can execute an I/O request in either a procedure-based manner or by using system threads. Procedure-based calls to the function processor allow the function processor to complete execution within the calling thread. System threads provide function processors with the mechanism to do extended processing, including I/O waiting, after returning control to the user thread. System threads belong to the function processors that queue requests to them.
8.2.2 Objects Used by the I/O System

The Mica I/O architecture defines three I/O objects:

- Function Processor Unit (FPU) object
- Channel object
- Function Processor Descriptor (FPD) object

The functions of these objects are described in the following sections. The I/O architecture defines two significant data structures that are not owned by any particular object. These two structures are the I/O request packet (IRP) and the I/O status block (IOSB).

The IRP maintains the user's I/O request, as well as some bookkeeping information that is used by various components and objects in the I/O system. The IRP is allocated when an I/O request is made and is deallocated when the request is completed.

The IOSB status block contains the final status of the I/O request and other data (such as byte transfer count) that is written to it when the I/O request completes.

8.2.2.1 FPU Object

A function processor accepts requests on one or more function processor units (FPUs). An FPU represents a particular resource to higher levels of software. All requests to a resource are directed to its respective FPU, which then specifies the appropriate function processor to process the request. Examples of these FPUs are the ODS II unit, shadow unit, striping unit, device unit, MSCP unit, and so on.

8.2.2.2 Channel Object

A channel object describes a logical I/O path to an FPU on which I/O requests can be issued. The channel object receiving the initial user request maintains a listhead of all outstanding IRPs. This listhead is only used in the event that all outstanding requests on this channel need to be canceled. Channel objects are only associated with FPU objects and thread objects.

8.2.2.3 FPD Object

The function processor descriptor (FPD) object maintains the addresses of each global procedure in the function processor. The I/O architecture has a defined set of procedures that are common to all function processors. When the function processor is needed to process an I/O request, the address of the appropriate function processor procedure is looked up via the FPD object.

8.2.3 I/O Request Synchronization

The Mica I/O architecture supports two types of I/O requests:

- Synchronous
- Asynchronous

If a synchronous request is specified, the issuing thread is blocked until the request completes. If the request is asynchronous, then the issuing thread is not blocked, but continues to execute. The program issuing an asynchronous I/O request has the choice of specifying an AST procedure, an event object, or both to synchronize its execution with the completion of the request. When an asynchronous I/O request completes, the specified event object is signaled and/or the specified AST is queued.

8–2 I/O ARCHITECTURE
8.2.4 I/O Service Routines

The I/O Architecture specifies a set of well-defined interfaces to the I/O system for the purpose of initiating, canceling, and synchronizing I/O requests; as well as for creating, manipulating, and deleting of objects. Some of these interfaces to the I/O system are available via system service routines. Other interfaces are designated as internal, and are only available to components of the I/O system, such as function processor and system threads.

The diagram in Figure 8-1 shows a typical configuration of the I/O system. Before the I/O system can be used, an FPD object must be created for each function processor and an FPU object must be created for each available resource. After the system has been set up, the user can then create a channel object to an FPU, and issue an I/O request via the Request_IO system service routine. The user is notified when the request has been satisfied.

Figure 8-1: Overview of Mica's I/O Architecture
8.3 Functional Description

This portion of the chapter describes the Mica I/O system at an operational level. More implementation-specific information is provided in Section 8.4.

8.3.1 System Services, Executive Services, and Procedures

The I/O architecture defines two levels of services:

- System services
- Executive services

System services, which are preceded by an exec$, can only be executed from user-mode. The executive services, which are preceded by an e$, can only be executed from kernel mode, and are typically only invoked internally by various components in the I/O system.

All exec$ system services have a corresponding e$ executive service. These e$ executive services may call procedures defined in a function processor, or by the I/O architecture. I/O architecture (io$) procedures are typically called by the I/O architecture itself, and are not directly accessible through the services.

Figure 8–2 illustrates the inter-relationships between the system services, executive services, I/O architecture procedures, and function processor procedures in the Mica I/O architecture. Tables 8–1, 8–2, 8–3, and 8–4 briefly describe each service and procedure.

For more information on system services and executive services, see Chapter 9, System Service Architecture.
Figure 8–2: Relationships between Services and Procedures in the I/O Architecture

**System Services**
- exec$configure_fp
- exec$create_fpu
- exec$get_fpu_information
- exec$create_channel
- exec$get_channel_information
- exec$synchronize_io
- exec$cancel_io
- exec$request_io

**Executive Services**
- e$configure_fp
- e$create_fpu
- e$get_fpu_information
- e$create_channel
- e$get_channel_information
- e$synchronize_io
- e$cancel_io
- e$request_io

**Function Processor Procedures**
- Initialize
  - Configure FP
  - Initialize FPU
  - Get FPU Information
  - Cancel IO
  - Unload FPD
  - Execute IO
  - Complete IO
  - Synchronous IO Call
  - Remove FPU
  - Delete FPU
  - Initialize IO Parameters

**I/O Architecture Procedures**
- io$remove_tpd
- io$delete_fpu
- io$remove_tpu
- io$delete_fpu
- io$initialize_io_parameters
- io$remove_channel
- io$delete_channel
Table 8-1: I/O System Services

**FPD system services**

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>exec$configure_fp</td>
<td>Configure or deconfigure a function processor</td>
</tr>
</tbody>
</table>

**FPU system service routines**

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>exec$create_fpu</td>
<td>Create an FPU object</td>
</tr>
<tr>
<td>exec$get_fpu_information</td>
<td>Retrieve information about an FPU</td>
</tr>
</tbody>
</table>

**Channel system service routines**

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>exec$create_channel</td>
<td>Create a channel object</td>
</tr>
<tr>
<td>exec$get_channel_information</td>
<td>Retrieve information about a channel</td>
</tr>
<tr>
<td>exec$cancel_io</td>
<td>Cancel I/O request</td>
</tr>
<tr>
<td>exec$request_io</td>
<td>Initiate an I/O operation</td>
</tr>
<tr>
<td>exec$synch_channel_with_fpu</td>
<td>Synchronizes the channel's sequence number with the FPU's sequence number</td>
</tr>
<tr>
<td>exec$synchronize_io</td>
<td>Synchronize with completion of a previously-issued I/O request</td>
</tr>
</tbody>
</table>

Table 8-2: I/O Executive Services

**FPD Executive Services**

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>e$create_fpd</td>
<td>Create FPD object</td>
</tr>
<tr>
<td>e$configure_fp</td>
<td>Executive form of exec$configure_fp</td>
</tr>
<tr>
<td>e$unload_fpd</td>
<td>Perform cleanup related to FPD before deactivating function processor</td>
</tr>
</tbody>
</table>

**FPU Executive Services**

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>e$create_fpu</td>
<td>Executive form of exec$create_fpu</td>
</tr>
<tr>
<td>e$get_fpu_information</td>
<td>Executive form of exec$get_fpu_information</td>
</tr>
<tr>
<td>e$bind_fpus</td>
<td>Bind FPU objects together</td>
</tr>
<tr>
<td>e$unbind_fpus</td>
<td>Unbind previously bound FPU objects</td>
</tr>
<tr>
<td>e$register_ast_state_change</td>
<td>Registers a thread on an FPU</td>
</tr>
<tr>
<td>e$deregister_ast_state_change</td>
<td>De-registers a thread on an FPU</td>
</tr>
<tr>
<td>e$change_fpu_state</td>
<td>Changes the state of an FPU object</td>
</tr>
<tr>
<td>e$verify_io_access</td>
<td>Verifies access to an I/O object</td>
</tr>
</tbody>
</table>
### Table 8-2 (Cont.): I/O Executive Services

<table>
<thead>
<tr>
<th>Service</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>e$create_channel</td>
<td>Executive form of exec$create_channel</td>
</tr>
<tr>
<td>e$set_fpu_irp_size</td>
<td>Sets an FPU's IRP size</td>
</tr>
<tr>
<td>e$get_channel_information</td>
<td>Executive form of exec$get_channel_information</td>
</tr>
<tr>
<td>e$request_io</td>
<td>Executive form of exec$request_io</td>
</tr>
<tr>
<td>e$request_io_trusted</td>
<td>A trusted executive form of exec$request_io; less parameter checking is done</td>
</tr>
<tr>
<td>e$execute_io</td>
<td>Perform requested I/O operation</td>
</tr>
<tr>
<td>e$remove_callback_table</td>
<td>Removes callback table from channel</td>
</tr>
<tr>
<td>e$set_callback_table</td>
<td>Sets callback table in channel</td>
</tr>
<tr>
<td>e$set_channel_access</td>
<td>Sets access flag in channel</td>
</tr>
<tr>
<td>e$clear_channel_access</td>
<td>Clears access flag and granted access types set in channel</td>
</tr>
<tr>
<td>e$synch_channel_with_fpu</td>
<td>Executive form of exec$synch_channel_with_fpu</td>
</tr>
<tr>
<td>e$synchronous_io_call</td>
<td>Special internal service used for special interfaces between function processors</td>
</tr>
</tbody>
</table>

### IRP Services

<table>
<thead>
<tr>
<th>Service</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>e$allocate_fp_parameter_record</td>
<td>Allocate a fixed amount of space in IRP for FP parameter record</td>
</tr>
<tr>
<td>e$deallocate_fp_parameter_record</td>
<td>Deallocate space in IRP for FP parameter record</td>
</tr>
<tr>
<td>e$initialize_irq</td>
<td>Initialize I/O request packet</td>
</tr>
<tr>
<td>e$extend_fp_parameter_record</td>
<td>Extends the current FP parameter record</td>
</tr>
<tr>
<td>e$allocate_irq</td>
<td>Allocate an I/O request packet</td>
</tr>
<tr>
<td>e$deallocate_irq</td>
<td>Deallocate an I/O request packet</td>
</tr>
<tr>
<td>e$get_size_of_htl</td>
<td>Calculates the size of an HTL</td>
</tr>
</tbody>
</table>

### Synchronizing I/O Services

<table>
<thead>
<tr>
<th>Service</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>e$synchronize_io</td>
<td>Executive form of exec$synchronize_io</td>
</tr>
<tr>
<td>e$complete_io</td>
<td>Completes an I/O request</td>
</tr>
<tr>
<td>e$lock_io_buffer</td>
<td>Lock I/O buffers</td>
</tr>
<tr>
<td>e$unlock_io_buffers</td>
<td>Unlock I/O buffers</td>
</tr>
</tbody>
</table>

### Cancel I/O Request Services

<table>
<thead>
<tr>
<th>Service</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>e$cancel_io</td>
<td>Executive form of exec$cancel_io</td>
</tr>
<tr>
<td>e$cancel_io_by_thread</td>
<td>Cancel all outstanding I/O requests related to a thread</td>
</tr>
<tr>
<td>e$cancel_io_by_channel</td>
<td>Cancel all outstanding I/O requests for the issuing thread on a channel</td>
</tr>
<tr>
<td>e$insert_cancel_procedure</td>
<td>Inserts the address of an FP cancel procedure into an IRP</td>
</tr>
<tr>
<td>e$remove_cancel_procedure</td>
<td>Clears the cancel procedure address in an IRP</td>
</tr>
</tbody>
</table>
## Table 8-3: I/O Architecture Procedures

<table>
<thead>
<tr>
<th>Procedure</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>io$remove_fpd</td>
<td>Deactivate function processor prior to deleting FPD object</td>
</tr>
<tr>
<td>io$delete_fpd</td>
<td>Prepare an FPD object for deletion</td>
</tr>
<tr>
<td>io$remove_fpu</td>
<td>Call remove_FPU procedure in function processor</td>
</tr>
<tr>
<td>io$delete_fpu</td>
<td>Call delete_FPU procedure in function processor</td>
</tr>
<tr>
<td>io$state_change_ast_cleanup</td>
<td>Cleans up the state change AST's data structure</td>
</tr>
<tr>
<td>io$remove_channel</td>
<td>Inhibit further access to this channel</td>
</tr>
<tr>
<td>io$delete_channel</td>
<td>Prepare the channel for deletion</td>
</tr>
<tr>
<td>io uninitialized io parameters</td>
<td>Initialize parameters passed by exec$request io trusted</td>
</tr>
</tbody>
</table>

## Table 8-4: Function Processor Procedures

<table>
<thead>
<tr>
<th>Procedures</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>initialize io parameters</td>
<td>Initialize I/O parameters. This procedure verifies the parameters of a new request, allocates an IRP, and initializes the IRP with these parameters.</td>
</tr>
<tr>
<td>execute io</td>
<td>Perform I/O operation. This procedure passes a new or existing I/O request packet to a function processor.</td>
</tr>
<tr>
<td>synchronous io call</td>
<td>Synchronous I/O call. This procedure supports short, synchronous I/O requests without creating an IRP or causing a context switch.</td>
</tr>
<tr>
<td>initialize fpu</td>
<td>This procedure initializes a newly created FPU object.</td>
</tr>
<tr>
<td>remove fpu</td>
<td>Remove an FPU. This procedure causes a function processor to rundown reference pointers to the FPU object.</td>
</tr>
<tr>
<td>delete fpu</td>
<td>Delete an FPU. This procedure prepares an FPU object for deletion</td>
</tr>
<tr>
<td>get_fpu information</td>
<td>Get information on an FPU. This procedure retrieves information related to an FPU object.</td>
</tr>
<tr>
<td>unload fdp</td>
<td>Unload an FPD. This procedure performs any cleanup steps related to the FPD before the function processor image is deactivated.</td>
</tr>
<tr>
<td>configure fp</td>
<td>Configure function processor. This procedure creates the initial FPUs when the function processor is configured.</td>
</tr>
<tr>
<td>initialize</td>
<td>Initialize. This procedure is the first entry point into the function processor. Initialize is an internal procedure that is called after the function processor image is loaded. This procedure performs all necessary initialization, such as the creation of an FPD object, system threads, and any FPU objects that may be needed at this time.</td>
</tr>
<tr>
<td>complete io</td>
<td>Complete an I/O request. This is an internal procedure that performs whatever processing necessary to complete the request for this function processor.</td>
</tr>
<tr>
<td>cancel io</td>
<td>Cancel an I/O request. This is an internal procedure that allows the current function processor to force completion of its IRP.</td>
</tr>
</tbody>
</table>
8.3.2 Function Processors

A function processor is a collection of kernel-mode procedures and threads that execute I/O requests. A high degree of modularity is achieved by isolating all of the code relating to a particular type of I/O function to a single function processor. For example, the code that supports the file system's directory structure and the code that drives a disk device make up separate function processors.

The operations of the individual function processors are coordinated by the executive, to which all function processors share a common interface. In this scheme, the executive is free to operate as a generic, object-manipulating environment that does not have to be concerned with function-specific tasks.

Function processors are the executors of I/O requests dispatched by the service calls introduced in Section 8.3.1. Depending on the nature of the I/O request, a function processor may process and complete the request itself, or it may do some processing and pass the request off to another function processor. The mechanisms used to process an I/O request are described in Section 8.3.5.

Function processors have essentially two modes of execution: procedure-based execution in the context of the issuing thread, and execution within system threads owned by the function processor. Some function processors also execute code via callbacks from system threads in lower-level function processors, or calls from the system during I/O completion.

8.3.2.1 Function Processor Types

There are two function processor types:
- Virtual function processors
- Device function processors

Virtual function processors implement the logical layers of the I/O system. This type of function processor is used to implement all of the virtual-level operations, such as the file system, disk striping, virtual terminal support, and so on.

The primary task of a device function processor is to transport commands from a virtual function processor to the device hardware. The device function processor provides the same support as the device driver provides in VMS.

8.3.2.2 Interface Classes

Function processors sharing similar access characteristics are said to belong to the same interface class. All of the function processors in a class make up a single programming interface. Examples of common interface classes are the disk file system interface class, logical block class, logical magtape class, and so on.

An interface class is defined by its Request I/O function codes, I/O parameter record format, I/O status codes, and an expected set of semantics for each function. All function processors within the same interface class support the same function codes, I/O parameter record format, and FP parameter record format. (I/O parameter records are discussed in Section 8.3.3.1.1, and FP parameter records are discussed in Section 8.4.1.1.2.)

Some interface classes are designed to be supported by multiple function processors, such as the logical block interface class. Other interface classes may only be supported by a single function processor, which defines its own unique interface class.

The interface class supported by a function processor is encoded in a field in its function processor descriptor (FPD) object. There are separate ranges for Digital-defined and user-defined interface class codes. The definitions of Digital-defined interface classes are designed to change as little as possible. The addition and deletion of function codes should be avoided, and changes in I/O parameter record definitions must be minimized. Changes such as these should only be supported through parameter record item lists.
When planning a new function processor, it must be decided if the function processor is to support an existing interface class, or whether a new one is required. If an existing class is to be used, then the function processor must be written to accept all defined functions and to perform the expected semantics for each function. If a new interface class is defined, then the function codes, parameter records, and semantics for each function must be defined.

8.3.2.3 Function Processor Unit

As previously discussed in the overview of this chapter, each device, volume, or other type of unit supported by a function processor has a function processor unit (FPU) associated with it. An FPU represents a particular unit or volume (referred to here as a resource) to higher levels of software. All I/O requests to a resource are directed to its respective FPU.

Each FPU consists of a common header and a function-specific data structure that represents its respective resource. This function-specific data structure provides the same support as the volume control block (VCB) or the unit control block (UCB) data structures in VMS.

8.3.2.3.1 FPU States

An FPU can be in one of five states:

- **OFFLINE**—The corresponding unit is not currently known/reachable by the system.
- **AVAILABLE**—The FPU is believed to be available for use, but has not yet been readied to the ONLINE state.
- **ONLINE**—The FPU is ready for normal operation.
- **TRANSITION**—The FPU is currently in transition. Without any further intervention, the FPU changes to one of the other states on its own. However, in general, it is impossible to predict which state it will change to.
- **MAINTENANCE**—This state is like the online state with additional function-processor-specific diagnostics enabled.

The function codes for the FPU states are listed in Table 8-5.

<table>
<thead>
<tr>
<th>Table 8–5: FPU States Codes</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>io$sc_fpu_state_offline</td>
<td>FPU offline state</td>
</tr>
<tr>
<td>io$sc_fpu_state_available</td>
<td>FPU available state</td>
</tr>
<tr>
<td>io$sc_fpu_state_online</td>
<td>FPU online state</td>
</tr>
<tr>
<td>io$sc_fpu_state_transition</td>
<td>FPU transition state</td>
</tr>
<tr>
<td>io$sc_fpu_state_maintenance</td>
<td>FPU maintenance state</td>
</tr>
</tbody>
</table>

An FPU may be waited on. A wait is satisfied when the FPU is in any state but TRANSITION.

Note that FPUs that are placed in the OFFLINE or MAINTENANCE state must cancel or complete all current and outstanding I/O requests as soon as possible. An FPU must never get into a state where outstanding I/O requests in its internal queue cannot be processed or canceled.

8–10 I/O ARCHITECTURE
8.3.2.3.2 FPU Sequence Number

An FPU sequence number is used to handle a situation in which a lower-level FPU has a state change while an arbitrary number of requests may be outstanding. When a channel is assigned to an FPU that is not in the TRANSITION state, the sequence number from the FPU is stored in the channel object. Each time a request is issued or passed on via that channel, the FPU sequence number is copied into the IRP.

When an FPU changes state, its sequence number also changes. From now on, any request the function processor receives that has the outdated FPU sequence number in the IRP will be rejected by the function processor with an error.

For example, if a higher-level FPU passes a request over a previously existing channel, this request may be rejected. The higher-level function processor must detect that a state change has occurred, determine whether or not it wishes to continue to communicate with the lower-level FPU (such as after volume verification), and update its channel sequence number. The channel sequence number may be synchronized with the FPU sequence number by calling exec$sync_channel_with_fpu.

Note that, even when a straggling request is issued before the transition is held up for a reasonable amount of time, the lower-level FPU can still detect this condition and reject the request.
8.3.2.3.3 FPU State Change AST

State change ASTs are either user-mode or kernel-mode ASTs used to notify a thread that an FPU has changed state. FPU states are described in Section 8.3.2.3.1. Figure 8-3 shows the possible states an FPU object may be in, and the relationship between these states. Before a thread can receive an AST resulting from an FPU object state change, it must first register with the target FPU. Once a thread has registered itself on an FPU, it continues to receive an AST whenever the FPU object state changes. A thread may register with more than one FPU object. However, if it registers with the same FPU object more than once, only the most recent registration is used. A thread that is currently registered on an FPU may cancel its registration at any time. When the AST state change registration is canceled, the previously registered thread no longer receives state change ASTs. However, this does not prevent other threads registered on this FPU object from receiving a state change AST.

As stated earlier, in order for a thread to receive a state change AST, it must first register itself on an FPU. The thread registers itself by issuing the exec$\texttt{request\_io} service and specifying the \texttt{io$s\textunderscore enable\textunderscore state\textunderscore change\_ast} function code, along with the AST procedure to be executed. This request may be issued using either the exec$\texttt{request\_io} or \texttt{e$s\textunderscore synchronous\_io\_call} service.

An AST parameter is used by the thread to identify which FPU object changed state. This AST parameter must be specified, regardless of whether the request is issued from user or kernel mode. When the AST is delivered, the AST parameter is returned to the user- or kernel-mode thread.

For example, the AST parameter may be a pointer to a data structure containing the channel ID, or a reference pointer to the FPU object. When the AST is delivered to the thread, the AST parameter points to this data structure. The thread can then use the channel ID or reference pointer in this data structure to issue an exec$\texttt{get\_fpu\_information} service to retrieve the current state of the FPU that changed state.

By having an AST parameter point to a data structure that contains the channel ID or reference pointer, the thread does not have to determine which FPU the state change AST is for. When the function processor receives the request, it calls the e$\texttt{register\_ast\_state\_change} service to register the thread on the specified FPU. The e$\texttt{register\_ast\_state\_change} service allocates a state change AST data structure, and initializes it. This data structure is then linked to the FPU. The thread is now registered on the specified FPU, and can now receive state change ASTs.

A function processor may change the state of its FPU by calling the e$\texttt{change\_fpu\_state} service. The parameters for this service are: a pointer to the FPU object, the new FPU state, and an optional sequence number. If the sequence number is not specified, the sequence number in the FPU object is incremented by one. Otherwise, the sequence number in the FPU is replaced with the number specified by e$\texttt{change\_fpu\_state}. (For more details on the e$\texttt{change\_fpu\_state} service, see Section 8.4.3.5.3.) When the e$\texttt{change\_fpu\_state} service is called, the state of the FPU object is changed, and all registered threads have a state change AST queued to them. As each state change data structure is processed by e$\texttt{change\_fpu\_state}, it is marked to indicate that a state change AST is queued to this thread. After the data structure is marked, the state change AST is queued along with a piggyback AST to the thread, as specified by this data structure.

One of the functions of the piggyback AST is to unmark the state change data structure. When this data structure is unmarked, the state change AST is no longer queued, and the data structure is available to handle another state change AST. The piggyback AST executes the io$s\textunderscore state\textunderscore change\_ast\_cleanup procedure when the state change AST is delivered to the thread by the kernel. After the piggyback AST procedure completes, the kernel delivers the normal AST to the thread, and the thread's AST procedure is executed. This procedure is responsible for reading the current state of the FPU. This way the thread has the most current state. If the state of the FPU object should change after the AST has been delivered, another state change AST is queued. However, if the state of the FPU object should change before the AST is delivered, no additional AST is queued. A thread may cancel its state change registration by calling the exec$\texttt{request\_io} service with the io$s\textunderscore disable\_state\textunderscore change\_ast function code.
When processing the `io$disable_state_change_ast` function code, the function processor cancels the thread's state change registration by calling `e$deregistered_ast_state_change`. After this, the thread cannot receive an AST resulting from an FPU object changing states. See Section 8.4.3.5.2 for more information.

When an FPU object is removed (that is, when all of the object IDs for this FPU object have been deleted), then any threads that are still registered on this FPU are deregistered. See Section 8.4.3.6.1 for more information.

### 8.3.3 Requesting I/O

All I/O requests from user mode are made through the `exec$request_io` system service; all I/O requests from kernel mode are made through the `e$request_io` or `e$request_io_trusted` executive service.

The parameters used by `exec$request_io` or `e$request_io` can be summarized as follows:

- Channel object ID
- Function code describing the desired I/O function
- Completion parameters (I/O status block, event object, AST procedure, and AST parameter)
- I/O parameters specific to both the function code and the function processor class for the channel

These parameters correspond very well to those used by `sys$qio(w)` in VMS. The main difference between `exec$request_io` and `sys$qio(w)` is that `exec$request_io` passes the I/O parameters by means of an I/O parameter record. I/O parameter records are discussed in Section 8.3.3.1.1.

### 8.3.3.1 User-Defined Data Structures

Before an `exec$request_io` call is made, the user must declare an I/O parameter record to specify the interface class, record type, and the device-specific parameters to be passed on to the function processor. The user may also declare an I/O status block (IOSB) to receive the completion status of the I/O request.

### 8.3.3.1.1 I/O Parameter Record

All I/O parameters are passed by an I/O parameter record, which is defined specifically for the function processor class and function code of the request. This record contains a combination of the parameters themselves, pointers to parameters, and buffer descriptors.

In most cases, the I/O parameter record describes all possible parameters directly, providing a simpler interface than item lists—especially for programs in simpler languages, such as FORTRAN. However, when the total number of possible parameters is large and/or one or more parameters may appear an indefinite number of times, the record may use the standard item list pointer in the record header to point to an item list containing additional parameters (See Figure 8-4). In addition, this item list may be employed to allow the interface to be expanded from its initial definition simply by adding new item codes, as will be done in subsequent software releases.

All I/O parameter records consist of a standard record header followed by the body of the parameter record. Figure 8-4 illustrates a parameter record, showing the fields in the standard header.

The format of the I/O parameter record is as follows:

```plaintext
exec$io_parameter_record_header: RECORD
    record_type: word;
    interface_class: word;
    item_list: POINTER exec$item_list;
    version_number: longword;
END RECORD
```
The fields of the I/O parameter record are as follows:

- The *record type* is a code defined by the interface, that determines which of the record formats accepted by the interface is being specified. This is a sanity check. In general, the record type must be consistent with the function code specified in the call to exec$\text{request}_\text{io}.

- The *interface class* field specifies the function processor interface the request is directed to. This allows a sanity check to make certain that requests are only directed to the function processors they were intended for.

- The *item list pointer* is used to point to a standard item list, with some items being defined with the initial record definition (if necessary), and other items being defined in the future to expand the interface.

- The *body* of the record contains the parameters needed to perform the requested function. The body varies by function and by interface class.

- The *version number* is used to track changes to the record format.

### 8.3.3.1.2 I/O Status Block

When a request completes successfully, the exec$\text{request}_\text{io} service returns status information in an I/O status block (IOSB). The format of this block is the same as the IOSB in VMS, only each field is double in size, as shown in Figure 8–5. Sixteen bits (as in VMS) are no longer adequate for the *condition_value* and *byte_count* fields.

The format of an IOSB is as follows:

```plaintext
exec$iosb : RECORD
  condition_value : longword; ! I/O status
  byte_count : longword; ! I/O transfer count
  fp_condition : quadword; ! Filled in by the FP.
END RECORD;
```

The I/O status is returned by *condition_value*, which is described in Chapter 3, Status Codes and Messages.
8.3.3.2 Function Codes Used by exec$request_io

Most common function processors support the following exec$request_io functions: Get FPU Information, Access, Deaccess, Read, and Write. Most function processors support a number of other functions as well. However, only three I/O functions are standard across all function processors: Get FPU Information, FPU Access and Deaccess. These three function codes are discussed in the following subsections.

8.3.3.2.1 Get FPU Information Function Code

The Get FPU Information function code is the same for all function processors. The parameter record for the Get FPU Information function code is simply an item list, which is identical to that used in the exec$get_fpu_information service.

The interface_class field in the Get FPU Information parameter record contains a special null code. Some information is returned directly via pointers in the I/O parameter record for all function processors, but most information is returned via pointers in item list entries. Some of the possible item list entries are also common for all function processors. However, most item list entries are qualified by an interface class code, indicating that they belong to a single interface class.

The Get FPU Information function code serves two distinct purposes:

- Frequently a program issues an exec$get_fpu_information request on a newly created channel to determine which interface class it will be talking to on that channel, or to verify that the channel has been created to the expected interface class. For example, the Record Management Services (RMS) issues an exec$get_fpu_information request to determine which of its supported interface classes the channel is assigned to.

- Once a program knows, or assumes it knows, which interface class its channel is assigned to, it may issue a Get FPU Information request to retrieve class-specific information, such as terminal characteristics or File-11 volume information.

Note that most function processors also support some set FPU operations. However, there is no standard set FPU function code. Instead, set FPU operations are supported by the Get FPU Information procedure. When issuing set FPU operations through the Get FPU Information procedure, the caller must know exactly what function processor the call is being issued to.

Below is a table of generic item codes defined by the I/O architecture. These item codes must be supported by all function processors.
Table 8-6: Get FPU Information Item Codes

<table>
<thead>
<tr>
<th>io$C_item_interface_class</th>
<th>Integer</th>
<th>Specifies FPU interface class</th>
</tr>
</thead>
<tbody>
<tr>
<td>io$C_item_fpu_state</td>
<td>Enumerated type</td>
<td>Specifies FPU current state</td>
</tr>
<tr>
<td>io$C_item_fpu_bound</td>
<td>Boolean</td>
<td>Specifies FPU bound status</td>
</tr>
</tbody>
</table>

8.3.3.2.2 FPU Access Function Code

The FPU access function code (io$C_fpu_access) is the same for all function processors. This function code is used to specify the desired FPU access on a channel. Unless an FPU has been accessed by means of an FPU access I/O request, any other I/O request issued on this channel will fail.

The parameter record for the channel access request consists of a from_binder field and a set of access types desired by the caller, referred to as the desired_fpu_access set. The from_binder field is meaningful only if set by a kernel-mode thread. This field is used to determine if the FPU access I/O request originated from its binder (See Section 8.4.3.3.1 for more information on binding FPU objects). If a user-mode thread sets the from_binder field, the request is always rejected by the I/O architecture. The desired_fpu_access set is used to specify the desired FPU access types. If Mica security grants the access types requested for the FPU (this occurs when the function processor makes a call to es$verify_io_access when processing this function code), the granted access types are stored in the granted_access_type set in the channel object, and the request succeeds. For more information on FPU access types, see Section 8.3.9.1.1.

The format for the FPU access request parameter record is listed below:

```plaintext
TYPE
eexec$io_fpu_access: RECORD
    header: exec$io_parameter_record_header;
    from_binder: boolean;
    desired_fpu_access: SET [es$access_type];
    LAYOUT -
        header, from_binder, desired_fpu_access;
    END LAYOUT;
END RECORD;
```

FPU access requests may be issued at any time. Granted access types are not cumulative when issuing successive access requests. In other words, previously granted access types are cleared when a new access request succeeds. If the desired FPU access field in the record is empty, the function processor assumes that all valid FPU access types are desired.

All granted access types are removed when a deaccess request (an io$C_deaccess function code) is issued.

When a function processor processes an io$C_fpu_access function code, it calls the es$verify_io_access service. The es$verify_io_access service expects a channel pointer, a desired_access set, and a from_binder parameter. If the accesses can be granted, then the set of desired_access types are stored in the channel. If the accesses cannot be granted, then the set of granted_access_types stored in the channel object remains unchanged.
8.3.3.2.3 Deaccess Function Code

The Deaccess function code is the same for all function processors. However, the function processor parameter record (FP parameter record) for Deaccess does NOT have a standard definition. All function processors must accept Deaccess calls without an FP parameter record, and be able to complete a normal Deaccess without this record. The reason for this rule is that the executive reserves the right to call Deaccess on any channel it encounters during object rundown, and the executive will never specify an FP parameter record.

When processing the deaccess function code, the function processor calls the exec$clear_channel_access service. This service clears the access flag and the granted access types set in the channel.

During channel deletion, the executive issues a deaccess request to the function processor if the channel is still accessed. <LE>io$c_establish_callback <LE>io$c_get_channel_information The following function codes must be supported by all function processors:

- io$c_fpu_access
- io$c_deaccess
- io$c_get_fpu_information

8.3.3.3 Get Channel Information Function code

Table 8–7 lists the generic get channel information item codes used by the exec$get_channel_information service. These item codes must be supported by all function processors.

<table>
<thead>
<tr>
<th>Item Code</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>io$c_item_channel_access</td>
<td>Boolean</td>
<td>Specifies if channel is currently being accessed</td>
</tr>
<tr>
<td>io$c_item_granted_access</td>
<td>Enumerated type</td>
<td>Retrieve access types that has been granted to this channel</td>
</tr>
</tbody>
</table>

8.3.3.3.1 Set FPU Information Function Code

The io$c_set_fpu_information function code is the same for all function processors. Its parameter record is simply an item list, which is identical to the item list used by the exec$set_fpu_information service.

8.3.3.4 Request Synchronization

Four types of requests exist with respect to synchronization:

- Synchronous
- Event-signaled
- AST procedure notification
- Event-signaled/AST-procedure notification
8.3.3.4.1 Synchronous I/O Requests

By default, exec$request_io is a synchronous request. In this case, no user-specifiable event object or AST procedure is involved, and execution will not continue (except for AST execution, if ASTs are enabled) until the request is completed or canceled. The thread object contains a kernel event dispatcher object that provides this synchronization. This event dispatcher object cannot be set or cleared by normal calls.

For synchronous I/O calls, the executive synchronizes with the request directly during exec$request_io processing by making a call to the e$synchronize_io executive service and by specifying the kernel event object in the thread object. Note that the executive must synchronize via a call to e$synchronize_io, because the issuer could take an AST during the middle of a synchronous I/O wait, dismiss the AST without returning to the wait, and then issue another synchronous I/O. The second synchronous I/O must work properly, regardless of the fact that the previously issued synchronous I/O could prematurely set the event.

Due to the multithreading support in Mica, which is the preferred way to introduce parallelism in a process, synchronous I/O is the usual choice for user applications. User-level I/O support routines may wish to take advantage of the fact that the I/O status is available upon completion of a synchronous I/O operation, thus providing the possibility of signaling errors.

8.3.3.4.2 Asynchronous I/O Requests

The default synchronization can be overridden by specifying either an event object or an AST procedure (or both). If either of these are specified, execution always continues after the I/O service call, whether the request is complete or not. The final status of the I/O request must be read from the I/O status block when the request is complete.

If an event object is specified, the event is initially cleared during request initiation, then signaled when the request completes. By waiting on the event object, the program can synchronize with the completion of the request at any time after issuing the request.

If an AST procedure and AST parameter are specified, the AST procedure is called, with the AST parameter as input, when the request completes.

If both an event object and an AST procedure are specified, then, on completion of the request, first the event is signaled, then the AST procedure is invoked.

8.3.3.5 I/O Request Packets

An I/O request packet (IRP) is a data structure used internally by the I/O system to represent an individual request for I/O. An IRP is created by the I/O subystem when a request for I/O is issued, and remains in memory until the I/O operation completes. During the course of an I/O operation, an IRP may be passed from one function processor to another. Once a function processor passes an IRP to another function processor, it must no longer access that IRP.

An IRP consists of a fixed area and a free area. The IRP's fixed area contains request-independent parameters that are common to all IRPs, while the free area contains request-dependent FP parameter records that are specific to the I/O request. IRPs and FP parameter records are discussed in detail in Section 8.4.1.1.
8.3.4 Function Processor Parameter Processing

The format of the I/O parameter record for a request is determined entirely by the function processor. A function processor normally supports different I/O parameter records for different types of requests. The definition of the I/O parameter records is part of the function processor interface, and must be documented accordingly.

When dispatched to via the io$initialize_io_parameters procedure, the function processor is responsible for interpreting its input parameters and performing all of the validation necessary to ensure the integrity of the system. During this call, some parameters passed by the I/O parameter record are transformed to an internal representation.

Prior to calling the function processor via io$initialize_io_parameters, the executive checks if the device class code in the record matches the device class code of the function processor, and reports exec$c_wrong_device_class if there is no match.

A function processor should implement simple consistency checks through the record_type field of the standard I/O parameter record header. This helps the function processor to catch and properly report errors caused by passing the wrong record type. For such errors, the function processor should return exec$c_wrong_record_type.

Even if all of the header fields are correct, the function processor must make no assumptions about the validity of the parameter record, and it must be extremely defensive when processing the record. In general, the function processor must obey the following rules when processing parameters.

- All accesses to the parameter record, or to the user address space in general, must be protected by exception handlers. It is impossible to avoid the fact that any access to a user virtual address may result in an access violation. Even the second reference to the same field may trap. If an access violation does occur, the function processors exception handler should signal exec$c_access_violation.

- Prior to accessing a parameter from a call originating in user mode, the parameter must be probed to insure that the desired access (read or write) is allowed in user mode. Buffers must also be probed, even though they need not be accessed.

- All user parameters in the parameter record, or in other records pointed to by the parameter record, whose values are read and used during the processing of the request, must normally be "captured" (that is, copied to safe storage within the executive's address space) to the IRP. Typically, the only parameters not included in this rule are the contents of normal I/O buffers, although the buffer address and length must be captured. Once a parameter has been captured, all subsequent references to the parameter must be to the captured value, not to the actual user parameter.

- In general, for any field of $n$ bits in width, the function processor must assume that there are $2^n$ possible contents of that field. In other words, there must be no assumptions (subtle or otherwise) about the "reasonableness" of input data.

- Direct I/O buffers must be locked in memory via standard I/O architecture routines. (See Section 8.4.7.2 for a discussion on direct I/O buffers.) Buffered I/O buffers need not be locked in memory.

- If any error is detected during parameter processing, a condition should be raised. The function processor's own exception handlers must perform any cleanup that may be necessary for the initial processing of the request. This cleanup may include such things as deallocating allocated memory, unlocking locked buffers, releasing any locks or mutexes, and so on.
8.3.5 I/O Request Processing

This section discusses how I/O requests are handled by exec$request_io, e$request_io, and e$request_io_trusted. I/O requests handled by exec$request_io and e$request_io are checked by security to see if they are allowed. (For more information see Section 8.3.9.) For all requests, when the parameter processing in the function processor's initialize_io_parameters procedure is successfully completed, the function processor is called at the entry point to its execute_io procedure, which attempts to execute the requested I/O operation. If the function processor cannot complete the request itself, it may pass it on to another function processor (as discussed in Section 8.3.2). Once a function processor passes on an IRP, it must no longer access the IRP.

The rest of this discussion concentrates on requests the function processor decides to complete itself. To complete a request itself, the function processor may proceed in one of two ways:

- Complete the request while still running procedure-based in the context of the issuing thread, or while running procedure-based for the processing of a subsequent request.

- Queue the request for processing and completion by a system thread.

Even if the function processor decides to queue the request to a system thread, it may do a certain amount of processing while still procedure-based. The following subsections discuss procedure-based processing, general system thread processing, and threads driving hardware devices.

8.3.5.1 Procedure-Based Processing

Procedure-based processing begins when the function processor's execute_io procedure is called. At this point, the request is described entirely by the IRP, including the request-specific parameters filled in when the function processor was called at its initialize_io_parameters procedure.

While running procedure-based, function processors execute entirely within the context of the issuing thread. For process threads, all the buffers of the issuing process are directly accessible through their virtual addresses.

Any request may be completed directly during procedure-based execution. In fact, pseudo function processors, such as the message function processor, execute entirely procedure-based. Requests that cannot be completed immediately are queued internally for completion during the procedure-based processing of another request.

8.3.5.2 System Thread Processing

System threads provide function processors with the mechanism to do extended processing on a request, including I/O waiting, after returning control to the user thread. This allows the user thread to continue execution in parallel, and explicitly synchronize on its completion (via event or AST) at a later time. To initiate execution in a system thread, the function processor queues the IRP for the request to a listhead processed by a system thread.

System threads may not directly access any process buffers in an I/O request (See Chapter 4, The Kernel). However, the I/O interface does allow a system thread to issue I/O into a buffer (using physical addresses) specified in a user request, as long as the buffer is locked in memory.

One thing must be carefully considered for function processors that issue requests to themselves: If there is ever a case where a function processor's call to itself would also require a system thread, then there is a potential deadlock situation, regardless of how many system threads have been created. To prevent such deadlocks from occurring, the function processor must detect recursive calls from its own system threads, and not pass these requests off to another system thread. Instead, these requests should be processed directly by the original function processor thread. This can be implemented by calling the procedure directly to process the request, rather than queuing the IRP to a waiting thread to have it call that procedure.

8–20 I/O ARCHITECTURE
8.3.5.3 System Threads Driving Hardware Devices

Function processors may use system threads to drive hardware, such as I/O devices. The following sections discuss aspects of I/O processing specific to such threads.

8.3.5.3.1 Startup and Initialization

System threads driving actual hardware are created during FPU creation in the same manner as all function processor threads, as described in Section 8.4.3.2.2. Ideally, these threads would be created to run at the highest (or one of the highest) real time priorities. However, like all threads, they run at hardware IPL 0, except for brief periods of time, such as when initiating device I/O for a direct-connected device.

A function processor having to respond to hardware interrupts must create and initialize an interrupt object and connect it to the interrupt vector. This is normally part of FPU creation. Note, however, that there is not always a one-to-one relationship between threads and connected interrupts. A multi unit controller DFP, for example, could have one thread for each unit, but only one connected interrupt vector for the controller.

8.3.5.3.2 Creating the Thread

Before a function processor creates a thread, it must initialize an empty request queue, which is normally in the FPU. The thread is created to begin execution in an initialization procedure, and to enter its normal idle state waiting on the request queue.

8.3.5.3.3 Connecting to an Interrupt

Chapter 4, The Kernel, describes how an interrupt procedure is connected to an interrupt vector. When a function processor connects to an interrupt procedure, it must supply an argument to be passed for each call to the interrupt procedure. This argument directs the interrupt procedure to signal an event as the waking mechanism for a cooperating function processor thread. Typically, this argument is either a pointer to a kernel event object, or a data structure containing a pointer to a kernel event object.

Once established, the interrupt procedure is called once for each actual hardware interrupt that comes through the specified vector. Since interrupt procedures run at device IPLs, they should only do a very small amount of processing. If extended processing (or any processing that must be synchronized with threads) is required, the interrupt procedure may signal a function processor thread via the event object. This causes any thread waiting for this event to proceed.

Each time the interrupt procedure returns, an REI is executed in the kernel (after appropriate register restores).

8.3.5.3.4 Wait and Signal Operations

The control flow of function processor threads is normally implemented by Wait and Signal operations on the following kernel and executive object types:

- Queues
- Events (especially auto-clearing events)
- Mutexes
- FPU

A thread may wait on any combination of the above objects by means of a kernel wait procedure, k$s$wait$any. This wait procedure allows the thread to wait on one to some maximum number of objects, with an optional timeout value.
If any of the objects specified to the \texttt{k$wait\_any} procedure are already in a signaled state when the procedure is called, the thread continues immediately. Otherwise, the thread continues when one of the above objects is signaled, or the optional timer goes off.

In either case, an output of the \texttt{k$wait\_any} procedure is the number of the wait argument (or timeout) that was satisfied.

The following subsections explain in more detail how a thread might take advantage of these kernel object types. An additional subsection describes how function processor threads may synchronize with interrupt procedures and power failure.

8.3.5.3.5 Request Queue

The normal idle state of a system thread driving hardware is waiting for an IRP on a request queue. Insertion of an IRP into a request queue signals any waiting system thread.

In the procedure-based code of a function processor, an IRP may be queued to a request queue for processing by a system thread. If the system thread is already waiting, it will be unwaited. The system thread must then explicitly remove the IRP from the queue.

After completing one request, the system thread will normally loop back to again wait for another IRP on the queue. The \texttt{k$wait\_any} procedure is immediately satisfied when an IRP is put on the system thread's queue.

8.3.5.3.6 Events

Events are primarily used by interrupt procedures to signal threads. However, they may be used between threads, such as in the case where one thread signals an event for a thread in a higher-level function processor.

In the normal scheme of things, the thread first initiates some activity that causes the event to be signaled by the interrupt procedure, such as starting the associated device. It then specifies the event in a \texttt{k$wait\_any} call. When the event is signaled, the thread continues.

Threads normally use auto-clearing events. The initial state of an auto-clearing event is cleared. When a wait is satisfied on the event, its state is automatically cleared again to save the thread from having to explicitly do this later.

8.3.5.3.7 Mutexes

Some threads have a requirement to synchronize with other threads for critical code sections. One example is the per-unit threads supporting a multi-unit controller device. In this example, access to the common controller normally has to be serialized by a mutex.

The initial state of a mutex is signaled. Whenever a wait is honored on a mutex, the mutex is automatically cleared.

To synchronize with another thread, a given thread simply specifies a mutex in a \texttt{k$wait\_any} call. If the mutex is currently signaled, it is cleared, and the thread immediately continues. If the mutex is currently cleared, then the thread goes into a FIFO wait queue for the mutex. When the mutex is signaled, the first waiting thread in the queue is unwaited and the mutex is automatically cleared again. Whenever a thread proceeds from a \texttt{k$wait\_any} call as the result of gaining a mutex, the thread owns that mutex and must subsequently release the mutex by signaling it.

8–22 I/O ARCHITECTURE
8.3.5.3.8 FPUs

Some threads must synchronize with an FPU object as it leaves the TRANSITION state. A thread may wait on an FPU object by means of the e$wait_single or e$wait_multiple service. If the FPU object specified to the executive wait service is already in the signaled state (meaning that the FPU has just changed state), the thread continues to execute. Otherwise, the thread waits until it is signaled, or until the optional timer expires. The FPU object is signaled whenever the FPU changes to a state other than TRANSITION.

8.3.5.3.9 Synchronizing with Interrupt Procedures and Power Fail

In addition to using the standard synchronization objects described in the previous sections, a function processor thread driving a direct-connected device must also synchronize with any interrupt procedures for its device, as well as with possible power failures. In fact, when a thread wishes to initiate a hardware function, these two synchronization requirements typically overlap in time.

When a thread needs to manipulate device registers, or any data structure context that is also manipulated by an interrupt procedure, the thread must synchronize these manipulations with the execution of the interrupt procedure. For this purpose, the kernel routine, k$synchronize_execution, is used. (Note that it is inadequate to simply raise the hardware IPL, since the interrupt procedure may run on another processor in a multiprocessor system.)

In addition to the interrupt object for the interrupt procedure, the thread must specify an interrupt callback procedure and an argument for that procedure. The execution of the interrupt callback procedure is then synchronized with the interrupt procedure across all processors in the system. The interrupt callback procedure is also synchronized with the execution of any other callback attempts from other threads making a similar call with this interrupt object. Upon return from the interrupt callback procedure, the execution of the interrupt procedure is no longer blocked. The function processor must be coded so that all device register manipulation that must be synchronized occurs within such an interrupt callback procedure.

There are three things a function processor thread must do to properly handle and synchronize with power failure:

1. A thread must raise its priority to hardware IPL 7 when it wishes to lock out power failure for a very short period of time, such as for device register setup and I/O initiation. It does so by calling the kernel procedure, k$set_interrupt_priority_level. This call should always occur within the synchronized interrupt callback procedure described above.

2. A thread must create and initialize a power-up status object, and insert it in the kernel's power-up status queue. The status object allows the thread to detect a power failure during the window between the point where IPL was raised by the kernel to call the callback procedure, and the point where the IPL was raised to 7. To do so, the thread uses the kernel procedures: k$initialize_power_up_status and k$insert_power_up_status_queue. The thread must then test the status variable specified in the power-up status object immediately after raising to IPL 7. If the variable is TRUE, then the thread must abort device setup and unwind by signaling.

3. The thread must specify a power-up request object to cause it to execute a power-up AST before any other IPL 0 processing can occur when power is restored after a power failure. To do so, the thread must use the kernel procedures: k$initialize_power_up_request and k$insert_power_up_request_queue.

Assuming that a function processor thread is employing all of the above techniques, there are essentially three possible scenarios when a power failure occurs. These are listed below in increasing order of complexity:

1. The thread may be idle, waiting on its request queue. In this case, the thread need only worry about whether it has any power-up action to perform on the device, such as reinitialization. (The thread may also generate a signal to cause unwinding out of the AST procedure, possibly reinitializing the device as the result of handling the signal at non-AST state.)
2. The thread may be processing a request at IPL 0. In this case, the thread may or may not have initiated any device activity for this request prior to the power failure. Not only may the device have to be reinitialized, but the thread should normally generate a signal to cause the AST procedure and request processing to be unwound to a known point where a retry may be initiated.

3. The thread may be executing at device IPL in a callback procedure prior to raising to IPL 7. In this case, the power failure will occur, and the thread will resume execution upon power-up at device IPL. The power-up AST is now queued, but it is blocked by the elevated IPL. Here the thread must detect the power failure via the power status variable once IPL has been raised to 7, then exit the callback procedure without loading the device registers. When the callback procedure returns, the kernel drops the IPL back to 0. At this point, the AST occurs and processing proceeds exactly as in case 2 above.

All of the procedures mentioned in this section are documented in Chapter 4, The Kernel. Chapter 4, The Kernel also contains a coding example for starting a device with interrupt procedure and power fail synchronization.

8.3.6 Function Processor Callback

This section describes the mechanism used by a lower-level function processor to communicate with an upper-level function processor. This type of communication is referred to as function processor callback.

There are situations in which a lower-level function processor must communicate with an upper-level function processor. For example, the network interface (NI) function processor may receive a data packet from the network for which there was no outstanding request. In order to process this data packet, the NI function processor must be able to communicate this packet information with the router function processor above it.

In a normal configuration, an upper-level function processor communicates to a lower-level function processor through a channel object. The upper-level function processor issues either an e$\text{request\_io\_trusted}$, e$\text{execute\_io}$, or e$\text{synchronous\_io\_call}$ service call to pass a request to the function processor at the next lower level. One of the input parameters to these services is a pointer to the channel object, which contains the information needed to find the appropriate function processor.

The io$c\_\text{establish\_callback}$ function code is used to establish callback procedures between an upper-level function processor and a lower-level function processor. Among the parameters associated with this function code is a list of callback procedure entry points. Other parameters may be defined, as required by the individual function processors.

Function processors supporting io$c\_\text{establish\_callback}$ must follow the procedures described below:

1. An upper-level function processor passes its callback procedure entry points to the lower-level function processor whenever an upper-level function processor establishes a channel to the lower-level function processor.

2. When the lower-level function processor receives the callback entry points from the upper-level function processor, it creates a callback table in the channel object FPU data area. The size of this table may vary between function processors.

3. The lower-level function processor inserts the callback entry points into the callback table.

4. The lower-level function processor calls e$\text{establish\_fp\_callback}$ with the address of the callback table. The e$\text{establish\_fp\_callback}$ service stores the address of the callback table in the channel fixed area, and sets the channel access flag.

5. The lower-level function processor may store pointers to callback tables in the function-specific portion of the FPU. This allows the function processor to find the callback table(s) associated with each FPU object.

Figure 8–6 illustrates the process of establishing a callback table in a channel.
Figure 8-6: FP Callbacks

The lower-level FP creates a callback table in the channel object. The callback entry points are stored in this callback table.

Function processor calls `establish_fp_callback` to set a pointer in the channel object to point to the callback table.

The lower-level FP stores a pointer to the callback table in the FPU object for future reference.

FP stores a pointer to the callback table in FPU object.

Note that having the callback table stored in the channel object gives the function processor the flexibility of having different callback tables for each upper-level FPU. If a lower-level FPU has multiple sets of callback tables, then it may be desirable to build a data structure in the lower-level FPU's FP-specific area that maps an unsolicited external event to a particular callback table.
Also, if an upper-level function processor has multiple channels to the next lower-level FPU, the upper-level function processor may establish callbacks on some of the channels, or on all of the channels. However, if an upper-level function processor has established multiple callback tables for the same FPU, it is up to the lower-level function processor to manage these callback tables. Whenever a channel object containing a callback table is about to be deleted from the system, the I/O architecture issues a deaccess request (via e$request_io_trusted with the io$c_deaccess function code) to the lower-level FPU before the channel object is actually deleted. This gives the lower-level function processor a chance to remove the callback table pointer from the FPU object before the channel is deleted.

8.3.6.1 Performing Callbacks

Before a lower-level function processor can make a callback to a higher-level function processor, the lower-level function processor must process an io$c_establish_callback function code for the FPU that is to receive the callback request.

When an event requiring a callback occurs, the lower-level function processor uses its FPU to locate a callback table in the channel object of the upper-level function processor. The lower-level function processor locates the entry in the callback table that is associated with the event, which specifies the callback procedure to execute in the upper-level function processor.

If an FPU has multiple callback tables established, the lower-level function processor must determine which callback table to use. This is done by accessing a callback table array in a function-processor-defined data structure in the FPU. A callback table array contains pointers to all of the callback tables, as well as information identifying which callback tables to use for which events. For example, the NI function processor supports multiple callback tables. When a message is received on the NI, the network address in the message is used to determine which callback table to use.

When an event occurs, the lower-level function processor uses the callback table array in the FPU to locate the callback table associated with the event. Once the callback table is located, the function processor can then find the entry for the event and determine which callback procedure to execute in the upper-level function processor (See Figure 8-7).
8.3.7 I/O Completion

I/O requests are completed by calling the e$complete_io service. If an interrupt procedure detects that an I/O request has completed, it cannot complete the request directly. Instead, the interrupt procedure must signal an event to wake a thread to complete the request. Whenever a thread determines that an I/O request has completed, it completes the request by calling the e$complete_io service, which is an internal routine discussed later in Section 8.4.7.1.

8.3.8 I/O Cancellation

Outstanding I/O requests can be canceled via the channel object to which they are queued (either by an explicit user call, or by object rundown on a channel being deleted), or by the system via the thread object during thread rundown. There is no mechanism to cancel individual I/O requests from user mode.

Internally, requests are canceled individually by a cancellation procedure whose address has been stored in the cancel_procedure IRP field. This field is filled in by the function processor that owns the IRP and wishes to support cancellation over a given period of time.

The Cancel request itself is always completed immediately. However, it is still necessary for user and system software to synchronize with the original requests to know when they are done, since their cancellation will complete asynchronously. However, when the executive is running a thread down, all outstanding I/O request(s) are canceled synchronously.

See Section 8.4.8.1.1 for more information on cancellation.
8.3.9 I/O Security

The I/O architecture uses Mica security to provide protection for FPU and channel objects, which are protected by specifying an access control list (ACL) when the object is created. An ACL contains a list of one or more access control entries (ACEs). Within an ACE, there is a list of zero or more identifiers and an allowed_access set. This set is used to determine which access types can be granted if the right identifiers are held by the thread.

Each field in the allowed_access set corresponds to a particular access type. Mica security has defined a standard set of access types that are part of every access set. The remaining fields in the access set are for object-specific access types, which have been defined by the I/O architecture for both the channel and FPU objects. Access types for channel objects are used to control object deletion, changes to the ACL, getting channel information, and the right to issue I/O requests on a channel. The access types for FPU objects are used to restrict access to the FPU (which includes getting FPU information, FPU deletion, and so forth) and to control function code processing. See Section 8.3.9.1 for more information on protecting FPU objects.

8.3.9.1 Protecting FPU objects

An FPU object is protected by assigning an ACL to it. An ACL may be specified as an input parameter to the exec$create_fpu service. If an FPU object is created with an ACL, the ACL is linked to the FPU by the object architecture, after which access to the FPU is regulated by Mica security. (See Chapter 10, Security and Privileges for more information on how ACLs are used.)

The FPU's ACL can be used to restrict channel connections to an FPU, control FPU deletion, and restrict I/O requests issued to an FPU. The level of protection is defined by the contents of the ACL. An ACE defines which identifiers are associated with which access types. When an object is referenced, Mica security grants or denies access based on the identifiers held by the thread and the contents of the object's ACL.

Before an I/O request can be issued on a channel, the channel must first be accessed by an access I/O request. If the I/O request completes successfully, then granted access types are stored in the channel object. The channel is now ready to accept other types of I/O requests. Only I/O requests that are allowed by the access types stored in the channel's granted_access_types set are accepted by exec$request_io, all other requests are rejected. The granted_access_types set provides performance optimization to eliminate the need for an ACL check every time exec$request_io is executed.

8.3.9.1.1 FPU-Specific Access Types

The I/O architecture defines a set of FPU-specific access types for FPU objects. All access types listed below, with the exception of the FPU allow channel connection, have been defined to assign access types to function codes. (See the discussion on function code mapping later in this section.)

- FPU management access
  Use to grant access to FPU management function codes.

- FPU maintenance access
  Use to grant access to FPU maintenance function codes, such as io$c_xxxx_ready_fpu and io$xxxx_unready_fpu.

- FPU performance access
  Use for granting access to function codes related to performance monitoring.

- FPU diagnostic access
  Use for granting access to function codes related to running diagnostics.

- FPU allow channel connection access
  Use for granting channel connections to this FPU.

8–28 I/O ARCHITECTURE
- FPU get information access
  Use for granting access to the $c_get_fpu_information function code.
- FPU accounting access
  Use for granting access to function codes related to accounting, quotas, and so on.
- FPU access
  Use for granting access to access and deaccess function codes (for example, $c_fpu_access, $c_access, and $c_deaccess). This access type can be used to restrict the processing of access and deaccess function codes to those threads holding the identifier associated with this access type.

The FPU object-specific access types are defined below:

```
VALUE
  $c_access_management: $type_access_type = $c_specific_access_type_1;
  $c_access_maintenance: $type_access_type = $c_specific_access_type_2;
  $c_access_performance: $type_access_type = $c_specific_access_type_3;
  $c_access_diagnostic: $type_access_type = $c_specific_access_type_4;
  $c_access_allow_channel: $type_access_type = $c_specific_access_type_5;
  $c_access_get_information: $type_access_type = $c_specific_access_type_6;
  $c_access_accounting: $type_access_type = $c_specific_access_type_7;
  $c_access_access: $type_access_type = $c_specific_access_type_8;
```

Chapter 10, Security and Privileges, defines a set of standard access types that are applicable to all objects. Examples of these access types include: read access, write access, delete access, read/write access to the ACL, execute access, and so on.

**Defining Access Types for Function Codes**

Each interface class defines the access type(s) associated with each function code. A function code may have one or more access types associated with it. In order for a function code to be processed during an I/O request, all of the access types associated with the function code must be stored in the channel’s granted_access_types set. For each function processor, there is at least one function code access type (FCAT) table. The interface class defines the initial FCAT table, which is used by the I/O architecture to determine the access types for all legal function codes when I/O requests are issued to the object.

The FCAT table is indexed by function code value. Each element in the table corresponds to the access types needed for granting access to the function code. All function code access type tables are statically stored with the function processor image. A function processor may have more than one FCAT table. However, a function processor must have at least one table for FPU objects. This table is defined by the interface class.

Some function processors may have additional I/O objects, referred to as secondary objects. For each secondary I/O object, there must be a corresponding FCAT table. For example, the Files-11 function processor has a volume FPU object and a file object (which is its secondary I/O object). Therefore the Files-11 function processor must have an FCAT table for each I/O object. When the FPU FCAT table is created, a pointer to it is supplied to the $create_fpd service and stored in the FPD object, when created. When a channel is created, the pointer to the FPU FCAT table is retrieved from the FPD object and stored in the channel object.

The format of the function code access table is shown below:

```
TYPE
  $function_codes_access_table (number_of_function_codes: integer [0..]):
    RECORD
      access_types: array [number_of_function_codes]: SET [$access_types];
    END RECORD;
```
8.3.9.1.2 Restricting Function Codes Access

An FPU object can be protected so that certain types of I/O requests will fail during the execution of exec$request_i0. This is done by restricting the channel’s access to the I/O object. The access a channel has to an I/O object is determined by the set of granted_access_types types stored in the channel. I/O requests matching the access types in a channel’s granted_access_types set are allowed to pass through the channel.

Each function code in an interface class has one or more FPU access types or standard security access types associated with it. During the execution of exec$request_i0, the access type(s) of the function code is determined and checked against the granted_access_types set in the channel object. If all of the access types associated with a function code are in the channel’s granted_access_types set, then the request is accepted; otherwise it is rejected. If a function code has more than one access type, then all of these access types must be in the channel’s granted_access_types set before the request can be accepted.

Therefore, by controlling which access types are granted, the range of I/O requests that may be issued on a channel can be controlled. The FPU’s ACL defines which access types can be granted for a given thread.

8.3.9.1.3 Restricting Channel Connections to an FPU

Channel connections to an FPU object can be controlled by defining an ACE in the FPU’s ACL, so that the FPU allow channel connection access type is restricted. Threads that do not pass security cannot establish a channel to the FPU using the exec$create_channel service. The exec$create_channel service succeeds only if Mica security grants access. Even though an FPU object can be seen, or its object ID is accessible, channels cannot be created to it, unless Mica security grants access.

8.3.9.2 Protecting Channel Objects

A channel object can be protected by assigning an ACL to it. The ACL is specified as an input parameter to the exec$create_channel service. If an ACL was specified during channel creation, then access to the channel object is controlled by Mica security. (See Chapter 10, Security and Privileges for more information on how ACLs are used.)

8.3.9.2.1 Channel Specific Access Types

The I/O architecture defines a set of channel-specific access types for channel objects. These access types are used to control channel object deletion, getting channel information, and restricting threads that may issue I/O requests on a channel.

Below is a list of channel-specific access types:

- Channel Request I/O
  
  Use to grant request I/O access to a channel. If a thread does not have the identifier that grants this access, then all calls to exec$request_i0 will fail.

- Channel Get Information
  
  This access type, if granted, allows a thread to get channel information by either issuing an I/O request with the io$c_get_channel_information function, or by calling the exec$get_channel_information service.

- Channel Delete
  
  This access type is defined by Mica security, so it is not a channel-specific access type. Rather, it is listed here for reference purposes. Only user-mode threads that have been granted this access type are allowed to delete this channel when calling exec$delete_object_id.
The channel-specific access type is defined below:

```
VALUE
    ioSc_access_request_io:  e$access_type = eSc_specific_access_type_1;
    ioSc_access_get_information:  e$access_type = eSc_specific_access_type_2;
```

The access type definition for the channel delete access is defined in Chapter 10, Security and Privileges.

8.3.9.2.2 Accessing an I/O Object via a Channel

Before a channel can be used to issue I/O requests, it must first be accessed. This is done to inform the function processor how the channel is to be used. For example, a channel may be used to access an FPU object, file object, port object, and so forth. When an access is issued, the requested access types are only granted to the channel if the initiating thread is granted access by Mica security and the function processor. If the access request is granted, then the desired access types are stored in the channel's granted_access_types set. Unless an access request completes successfully, all I/O requests, other than another access request, will fail.

Figure 8–8 illustrates the steps for accessing an FPU:

- **Step 1**—An FPU access I/O request is issued using the exec$request_io services.
- **Step 2**—When the function processor receives the IRP containing the FPU access request, the request is verified by calling e$verify_io_access. If the desired access types specified by the channel access parameters are granted, then they are recorded in the channel object, and the request completes successfully.
- **Step 3**—The granted FPU allowed access types are stored in the channel object.
The set of granted_access_types in a channel can be changed by issuing another I/O access request. All granted access types in the channel's granted_access_types set are cleared when a deaccess I/O request is issued and completes successfully.

Not all I/O requests are for FPU objects. Some function processors have secondary I/O objects, such as file objects and port objects, to which an I/O request can be issued. For example, the Files-11 function processor allows I/O requests to be issued to a file object after it has been first accessed.

Below is an example of what happens when issuing a file access request:

1. File access I/O request is issued.
2. IRP is built and passed to the Files–11 function processor for processing.

3. The Files–11 function processor processes the file access request. If the file can be accessed, a file object is created along with its ACL, and a reference pointer to the file object is stored in the channel. If the requested file cannot be accessed, the request fails.

4. If the previous step succeeds, the thread’s access to the file object must be verified. The function processor does this by calling the esverify_io_access service. The function processor must supply the reference pointer to the object, the pointer to the FCAT table, the set of desired access types, and the channel reference pointer.

   • If the desired access types are granted, they are stored in the channel’s granted_access_types set, and the pointer to the FCAT table is stored in the channel header. The access request succeeds.

   • If the desired access types are not granted, then access to the object is not allowed, and the function processor must perform the steps necessary for cleanup. At this point the access request fails.

The granted_access_types set in a channel object is checked by the exec$requests/io service to filter out unauthorized I/O requests through the channel.

8.3.10 I/O System Configuration

The following subsections describe the configuration of both the physical and virtual layers of the I/O system.

8.3.10.1 Configuration of Virtual Layers

Once the physical FPUs are configured, Mica allows the construction of successive virtual abstractions (also represented by FPUs) on top of the physical devices. The best examples are with disk units, out of which stripe sets, shadow sets, and virtual volumes may be constructed.

Each FPU, or set of FPUs, are built by an iterative process from the bottom up, beginning with the actual physical device FPUs. Given one or more level n FPUs, a higher-level FPU (n+1) is created under the control of a configuration program as follows:

1. If level n FPU(s) are not already allocated, then allocate them.
2. Create channel(s) to the level n FPU(s).
3. Ready the level n FPU(s) (see Section 8.3.2.3.1 on FPU states).
4. Create, bind, and allocate the FPU for level n+1 in the following steps, passing in one or more level n FPUs as input:

   • Create level n+1 FPU, and initialize it.
   • Bind the level n+1 FPU to the level n FPU. FPU access types to be disabled can be specified as parameters to the bind service. The disallowed access types disable user functions for the level n FPU (see Section 8.4.3.3). This is done by calling the esbind_fpus service.
   • Create channel(s) to the level n FPU(s), and acquire pointers to these FPUs.
   • Delete channel to level n FPU(s).
   • Allocate level n+1 FPU.
   • Return object ID for level n+1 FPU to caller.
5. Delete channel(s) from Step 2.
6. If FPU(s) were allocated in Step 1, deallocate them.

8.3.10.2 Connection of Channels, FPUs, and FPDs

Figure 8–9 illustrates the connections between the various objects defined by the I/O architecture.

Figure 8–9: Connection of Channels, FPUs, and FPDs

8.3.11 Error Logging and Diagnostics

Some function processors implement error logging and device diagnostic support. The standard support for these areas is discussed in Chapter 23, Error Logging, and the individual chapters for the function processors.
8.4 Internal Design

This section of the chapter describes the internal structure of the I/O Architecture. The first part of this section describes the primary non-object data structures used by the I/O system. The remainder of this section describes the structure of each I/O system object, followed by the system services, executive services, and procedures used to control that object.

8.4.1 I/O Data Structures

This section presents the primary data structures of the I/O system. Some of the data structures referred to here are discussed elsewhere. The software process control block and the thread object are described in Chapter 6, Process Structure. File system structures, like the volume control block (VCB), are described in Chapter 25, Files-11 ODS2 Function Processor.

A brief description is given for each data structure, followed by subsections on linkages (how the structure is linked into the system), when the structure is allocated/deallocated, and what fields the structure contains. The description of the data structure fields are not meant to be bit-for-bit complete, but rather, to provide a better understanding of how those data structures are used.

8.4.1.1 The I/O Request Packet Structure

The I/O request packet (IRP) describes an active I/O request in the system issued by exec$request_io.

Linkages

All outstanding I/O requests are linked to pending I/O lists off of both the channel object and thread object for the request. These linkages are for bookkeeping purposes.

The IRP points to its respective software thread object and channel object.

The IRP also contains a utility link word for linking the IRP to internal queues, as desired by the function processor that currently owns the IRP.

Allocation

The IRP is allocated by the function processor when a valid call is made to exec$request_io, e$request_io, or e$request_io_trusted. It is deallocated when the request represented by the IRP is completed (the IRP is deallocated during the I/O completion phase).

Content

The IRP contains both request-independent information (common to all requests) and request-dependent information. The request-dependent information is in parameter records.

Figure 8-10 shows the Pillar definition for the fixed portion of the IRP.
Figure 8–10: I/O Request Packet Definition

```c
esio_request_packet (size_of_irp_free_area: integer[0..]) : RECORD
  CAPTURE size_of_irp_free_area;              ! Captured Size of fpu free area
  fpu_queue_link: esqueue_entry;             ! Used to link IRP into an FPU queue
  channel_list_link: eslinked_list;          ! Used to link IRP into channel list
  channel_object_pointer: POINTER eschannel_object_body;  ! Pointer to initial Channel obj
  thread_object_link: eslinked_list;         ! Used to link IRP into list in thread obj
  thread_tcb_pointer: POINTER esthread_control_block;  ! pointer to the TCB
  current_status: esiosb;                   ! Current status of this request
  iob_ptrn: POINTER esiosb;                 ! pointer to the IOSB
  user_event: POINTER esevent_object;       ! user event object for Async I/O
  kernel_event: POINTER kdispatcher_object (event);  ! kernel event object for async or syncI/O
  ast_procedure: POINTER esast_procedure;    ! ast procedure for async I/O
  ast_parameter: esunsigns;                 ! Parameter for ast procedure
  normal_ast: kcontrol_object (ast);        ! Normal Ast Object
  special_ast: kcontrol_object (ast);       ! Special kernel mode AST object
  cancel_procedure: escancel_io_procedure;  ! FP’s cancel procedure address
  cancel_issued: boolean;                   ! IRP cancel flag
  host_transfer_list: POINTER eshost_transfer_list;  ! Pointer to host transfer list
  in_page: boolean;                        ! paging I/O request flag
  release_channel: boolean;                ! Device Priority boost
  stay_in_current_thread: boolean;         ! Stay in the current thread’s context
  fpu_sequence_number: longword;            ! FPU’s sequence number
  current_fp_parameter_record: POINTER esfp_parameter_record;  ! Pointer to current FP params
  current_free_area_size: integer;         ! Size of free area
  free_pointer: POINTER quadrword;         ! Pointer to free area in the IRP
  free_area: quadrword_data(size_of_irp_free_area);  ! Start of free area.
END RECORD;
```

When first allocated and initialized, the IRP consists of a fixed area containing the request-independent parameters, and a large free area at the end for the allocation of request-dependent FP parameter records. The fixed area is discussed in the following subsections.

### 8.4.1.1.1 IRP Free Area

The IRP free area is managed as a LILO stack. It is an inverted stack with respect to normal use, since it grows in the direction of increasing addresses. Allocation is done by simply advancing the free_pointer. Deallocation is done by reversing the free_pointer in the direction of decreasing addresses.

One reason for having the IRP free area is that the function processor can allocate small blocks from this area without fragmenting system pool. These small blocks are used to maintain FP parameter records, which are written into the IRP by each function processor it is passed to.

When the exec$request_io service is invoked, the target function processor creates an FP parameter record by combining the parameters from the body of the I/O parameter record with a header containing information to be used upon completion of the request. The FP parameter record is then copied into the IRP’s free area.

Among the parameters in the IRP’s fixed area are the free_pointer (which points to the next free quadword in the free area) and the current_parameters pointer (which points to the FP parameter record created by the exec$request_io call). This is illustrated in Figure 8–11.
Figure 8-11: I/O Request Packet

If the IRP is to be passed to a second function processor, then a second FP parameter record is allocated. A back pointer in the second FP parameter record is filled in to point to the first FP parameter record, and current_parameters is filled in to point to the second FP parameter record. This sequence is repeated if the IRP is passed on to further function processors.

Procedures used to manipulate of the free area and current_parameters pointer are described in Section 8.4.5.

Figure 8-12 depicts an IRP that has been passed to a third function processor.
During I/O completion, the FP parameter records are used to locate their respective function processors to perform clean up operations. As lower-level function processors return from the I/O completion callback, their FP parameter records are returned in LIFO order to the IRP's free area. This is necessary, since higher-level function processors may decide to retry a failed request on another unit, or continue a successful request with a subsequent call.

In all cases, the function processor currently owning the IRP always has access to its parameters via the `current_parameters` pointer, after type-casting `current_parameters` as a pointer to the expected function-specific record type.

### 8.4.1.1.2 Function Processor Parameter Record

Function processor parameter records (FP parameter records) hold the user I/O parameters for the request in an internal format, or the parameters of an internal request passed from one function processor to another. In addition, an FP parameter record may hold a certain amount of internal context for the request.

All user I/O parameters must be captured at the time the request is issued, verified, then stored in an FP parameter record. These parameters are captured because it is possible for the issuing thread, or another thread, to modify or delete the address space after/while the function processor verifies the parameters. By capturing I/O parameters, an internal copy of the original parameters is made that cannot be compromised by any later modifications. In some cases, the parameters are stored in the IRP in a different format than that in which they were specified in a user call. For example, object IDs are converted to reference pointers, and buffers may be stored via host transfer lists (HTLs).

Included within an FP parameter record, the function processor may have subrecords pointed to by other fields in the FP parameter record. Pointers to these subrecords may be safely passed on to lower-level function processors.

For example, a function processor may decide to allocate an HTL from the free area and point to it from the I/O parameter area.

Another example is for buffered output. A function processor wishing to support buffered output may decide to either copy a small output buffer into the parameter area itself, or to separately allocate the buffer from non-paged pool, copy the data into this buffer, and store the address in the FP parameter record. In the latter case, it is necessary for the function processor to deallocate the buffer when its
completion procedure is called. (Buffered input is handled in the completion procedure, as described in Section 8.4.7.1.1.)

Function processors define their own FP parameter records for the IRP. For user-specified functions, this record typically stores the parameters of the user call in an internal format, along with other data fields required for the request. All FP parameter records must be prefixed with the header shown below, using LAYOUT in Pillar to fix the position at the front.

The FP parameter record header contains:

- A back pointer to any previous FP parameter records in the IRP
- The completion procedure to call when the request completes
- A call_on_error flag that determines if the function processor is to be called only on error, or on all completions
- The sequence number of the channel the request is passed on (this is filled in by the e$execute_io service)
- The size of the FP parameter record

The FP parameter record is defined as follows:

```
  e$fp_parameter_record  =  RECORD
      previous_fp_record:  POINTER  e$fp_parameter_record;
      completion_procedure:  POINTER  e$complete_io_procedure;
      sequence_number:  longword;
      call_on_error_only:  boolean;  ! Call FP on Error only
      record_size:  integer;  ! Size of fp parameter record
  END RECORD ;
```

The header contains standard fields that may be used to call the function processor in the issuer's context during the completion of the request. The use of these fields is discussed in Section 8.4.7.1.

### 8.4.1.1.3 Host Transfer List Data Structure

The host transfer list (HTL) is used to describe the direct I/O buffer. When a function processor is called at its initialize I/O parameters entry point, an HTL is created by the function processor, if direct I/O is to be done. The HTL contains a list link field, the buffer's starting virtual address, the length of the buffer, and a list of page frame numbers (PFNs) that map the buffer to physical pages in memory. This PFN list is used by memory management and the device. Memory management creates the PFN list initially when locking the buffer in memory; it also uses the PFN list to unlock the pages from memory at I/O completion. The device uses the PFN list to transfer data directly to or from memory.

The host transfer data structure is defined as follows:

```
  e$host_transfer_list  (number_of_pages:  integer  [0..])  =  RECORD
      capture  number_of_pages;  ! number of pages in buffer
      next_htl:  POINTER  e$host_transfer_list;  ! list link field
      starting_virtual_address:  POINTER  anytype;  ! start of buffer, virtual address
      length_of_buffer:  integer  [0..];  ! length of buffer
      process_control_block:  POINTER  e$process_control_block;  ! process control block
      offset:  integer  [0..$c_page_size-1];  ! buffer offset within
      page_frame_numbers:  ARRAY  [1..number_of_pages]  OF  e$spfn;  ! list of page frame #s
  END RECORD ;
```

The purpose of the list link field is to allow a single I/O request to specify multiple direct I/O buffers. This is accomplished by linking the HTLs together. Even though multiple buffers may be specified, they are locked down one at a time.
The process_control_block field of the HTL is used by memory management to facilitate probing and locking the buffer down. Not all buffers will have their starting address on a page boundary. Therefore, the offset field within the HTL is used to mark the start of a buffer within a page.

8.4.1.1.4 State Change AST Data Structure

The AST state change data structure is used by the I/O architecture to record which thread has registered with an FPU object to receive state change ASTs. This data structure is allocated from non-paged pool and is initialized when a thread registers to receive a state change AST. When the thread "de-registers" itself from receiving state change ASTs, the I/O architecture deallocates it. Also, when an FPU object is in the process of being deleted, any residual state change AST data structures are deallocated by the io$remove_fpu I/O architecture procedure after the function processor's Remove_fpu procedure is called. If the state change AST data structure is in use (currently queued) when the io$remove_fpu procedure attempts to deallocate the structure, the data structure is marked for deallocation, then deallocated when the state change piggybacked AST is run.

The format of the state change AST data structure is described below:

```plaintext
e$ast_state_change_record: RECORD
  mutex: k$dispatcher_object(mutex);
  flink: POINTER e$ast_state_change_record;
  blink: POINTER e$ast_state_change_record;
  fpu: POINTER e$fpu_object_body;
  thread: POINTER e$thread;
  mode: k$processor_mode;
  ast_routine: e$ast_procedure;
  user_ast_parameter: e$ast_parameter;
  ast_object: k$control_object(ast);
  ast_in_queue: boolean;
  deallocate_in_piggyback_ast: boolean;
END RECORD;
```

Arguments:

- **mutex**
  - Used to synchronize access to this structure
- **flink**
  - Forward link pointer to the next structure
- **blink**
  - Backward link pointer
- **fpu**
  - Reference pointer to fpu object
- **thread**
  - Reference pointer to thread object
- **mode**
  - User or kernel mode
- **ast_routine**
  - Address of the AST procedure to be executed
- **user_ast_parameter**
  - The user parameter to be delivered with the AST if the registration came from user mode
- **ast_object**
  - AST object
- **ast_in_queue**
  - If set, the AST object in this data structure has been inserted into the kernel AST queue
- **deallocate_in_piggyback_ast**
  - If set, this data structure will be deleted during the execution of the piggyback state change AST

8-40 I/O ARCHITECTURE
8.4.2 FPD Functional Interface and Design

This section details the structure of the FPD object, as well as the functional interface and design of each FPD service.

Listed below is a table of the available FPD object service routines.

<table>
<thead>
<tr>
<th>Table 8-8: FPD Object Services and I/O Architecture Procedures</th>
</tr>
</thead>
<tbody>
<tr>
<td>Service</td>
</tr>
<tr>
<td>--------</td>
</tr>
<tr>
<td>e$create_fpd</td>
</tr>
<tr>
<td>exec$configure_fp</td>
</tr>
<tr>
<td>e$unload_fpd</td>
</tr>
<tr>
<td>io$remove_fpd</td>
</tr>
<tr>
<td>io$delete_fpd</td>
</tr>
</tbody>
</table>

8.4.2.1 FPD Object Body Structure Definition

The FPD object describes a function processor. The FPD is referred to during I/O operations and is used for dispatching to the function processor.

Linkages

All FPD objects are in a system-level object container. The name of the FPD object is built from the name of the function processor.

Allocation

FPDs are created by the function processor during initialization, after its image is activated.

Content

The FPD contains the following fields:

1. A pointer to all of the function processor procedures.
2. The size of the function-processor-specific data structure to allocate with the FPU for this function processor.
3. The size of the function-processor-specific data structure to allocate with the channel object for this function processor.

Below is the definition of the FPD object body structure.

```plaintext
e$fpd_object_body: RECORD
  fpu_size: e$unsigned;
  channel_size: e$unsigned;
  fpd_procedures: e$fpd_procedures;
END RECORD;

e$fpd_procedures: RECORD
  e$unload_fpd_process: e$unload_fpd_procedure;
  initialize_fpu_proc: e$initialize_fpu_procedure;
  remove_fpu_proc: e$remove_fpu_procedure;
  delete_fpu_proc: e$delete_fpu_procedure;
  execute_io_proc: e$execute_io_procedure;
  get_fpu_information_proc: e$fp_information_proc;
  synchronous_io_call_proc: e$synchronous_io_call_proc;
  initialize_io_parameters_proc: e$init_io_parameters_proc;
  configure_fp_process: e$configure_fp_procedure;
END RECORD;
```
8.4.2.2 Creating FPD Object

An FPD object is created by the Create FPD (e$create_fpd) service. This service is an internal call issued by a function processor as part of its initialization procedure.

8.4.2.2.1 e$create_fpd

The e$create_fpd service is issued by the function processor itself during initialization. An FPD may be deleted via the e$delete_object_id call, which causes the function processor code to be unloaded.

The e$create_fpd call completes synchronously. The format of the call is as follows:

```
PROCEDURE e$create_fpd (  
  OUT fpd_id: e$object_id;  
  IN fpd_object_parameters: e$object_parameters;  
  IN fpu_size: e$unsigned;  
  IN channel_size: e$unsigned;  
  IN unload_fpd_proc: e$unload_fpd_procedure;  
  IN initialize_fpu_proc: e$initialize_fpu_procedure;  
  IN remove_fpu_proc: e$remove_fpu_procedure;  
  IN delete_fpu_proc: e$delete_fpu_procedure;  
  IN initialize_io_parameters_proc: e$init_io_parameters_proc;  
  IN execute_io_proc: e$execute_io_procedure;  
  IN configure_fpu_proc: e$configure_fpu_proc;  
  IN get_fpu_information_proc: e$fpu_information_proc;  
  IN synchronous_io_call_proc: e$synchronous_io_call_proc;  
  IN function_code_access_table: e$function_codes_access_table;  
) ;
EXTERNAL;
```

+++  
+  
+ Routine description:  
+  
+ Creates and initializes an FPD object.  
+  
+ Arguments:  
+  
+ fpd_id  
+   object id of resulting fpd object  
+ fpd_object_parameters  
+   parameters for the object architecture  
+ fpu_size  
+   size of fp specific area in fpu object  
+ channel_size  
+   size of fp specific area in channel  
+ unload_fpd_proc  
+   address of fp unload_fpd procedure  
+ create_fpu_proc  
+   address of fp create_fpu procedure  
+ delete_fpu_proc  
+   address of fp delete_fpu procedure  
+ initialize_io_parameters_proc  
+   address of fp initialize_io_parameters  
+ execute_io_proc  
+   address of fp execute_io procedure  
+ configure_fpu_proc  
+   address of configure_fpu procedure  
+ get_fpu_information_proc  
+   address of fp get information procedure  
+ set_fpu_information_proc  
+   address of fp set information procedure  
+ synchronous_io_call_proc  
+   address of fp synchronous io call  
+ function_code_access_table  
+   pointer to FPU function code access table  
+  
+ Return value:  
+  
+ none  
+  
+--

8-42 I/O ARCHITECTURE
The $create_fpd service routine does the following:

BEGIN $create_fpd
    call $create_and_initialize_object with appropriate parameters
    Calculate total FPU body size. This total size of the FPU body
    is the sum of the fpu_size input parameter plus the size of the fpu
    fixed header. Store this size in fpd object.
    load fpd object body with FP entry point address
    store size of channel object in FPD object body
    Insert object into the FPD object container
    Return
END $create_fpd

8.4.2.3 Configuring a Function Processor

A function processor is configured by the exec$configure_fp system service, which is typically called
by the function processor during initialization to configure the FPU objects.

The exec$configure_fp service is used to configure FPUs for a function processor, according to a pointer
to a record type defined by the function processor. This record type normally contains a description of
units to include or not include, along with physical parameters describing the units to be configured.
The function processor will configure the units by calling the create FPD procedure and specifying
its own FPD.

8.4.2.3.1 exec$configure_fp

The Configure Function Processor (exec$configure_fp) system service is used to configure or decon-
figure FPUs according to instructions provided by a record type defined by the function processor.
This record type normally contains a description of the units to include or not to include, along with
physical parameters describing the units to be configured.

Normally, the only software issuing this call is system configuration code, such as system management
programs. The format of exec$configure_fp is as follows:

PROCEDURE exec$configure_fp (  
    IN fpd_id: exec$object_id;
    IN function_code: integer;
    IN user_event: exec$object_id;
    IN fpd_parameters: POINTER anytype;
) RETURNS STATUS;
EXTERNAL;

++++
|   
|   Routine description:
|   
|   Configures and deconfigure a function processor.
|   
| Arguments:
|   
|   fpd_id object id of the fpd
|   function_code desired function
|   user_event object id of event to be signaled when done
|   fpd_parameters fpd configuration parameters.
|   
| Return value:
|   
|   TBD
|   
|--

I/O ARCHITECTURE 8–43
The `e$configure_fp` service routine does the following:

```
BEGIN e$configure_fp
  Probe parameters
  IF event object not valid THEN
    set status = Invalid event object id
  ELSE
    Get fpd reference pointer
    Point to fpd object body
    Get address of configure_fp from FPD object body
    Call configure_fp(fpd, function_code, event, item_list)
    Wait on event
    Set status = success
  ENDIF
END e$configure_fp
```

### 8.4.2.3.2 Function Processor's Configure FP Procedure

The configure FP procedure is called when the executive service `e$configure_fp` is executed. The purpose of this procedure is to allow the function processor to configure itself. The function processor may create system threads and FPU objects if they are required for configuration. When the configuration process is complete, the FPU object must be ready to accept a channel to it.

The executive type definition for the Configure FP procedure is as follows:

```plaintext
TYPE
e$configure_fp_procedure:
  PROCEDURE (IN function_code: integer;
             IN user_event: e$object_id;
             IN configure_fp_parameter: POINTER anytype CONFORM);
)
```

The function processor defines its Configure FP procedure as follows:

```plaintext
PROCEDURE configure_fp (
  function_code,
  user_event,
  configure_fpparameters
) OF TYPE e$configure_fp_procedure;
```

```plaintext
+++ 
! Routine description: 
! The function processor configure fpd interface definition 
! Arguments: 
! function_code function desired
! user_event object id of event to be signaled
! configure_fp_parameters configuration parameters
! 
! Return value: 
! None 
! ---
```

8-44 I/O ARCHITECTURE
8.4.2.4 Removing and Deleting FPD Objects

The owner of an FPD object deletes the FPD object by calling the exec$delete_object_id service defined by the object architecture. This service calls the remove and delete I/O architecture procedures for the object before it is actually deleted. These procedures are described in the following sections. For details on the exec$delete_object_id service, see (obj_chap).

8.4.2.4.1 io$remove_fpd

The io$remove_fpd service is called when all object IDs for the FPD object are deleted. The function processor is not called at this time.

Note that it is still valid for the function processor to store a pointer to its FPD in static storage during initialization without incrementing the pointer count, since the function processor code and static storage are unloaded before the FPD is deleted.

Although it is conceivable that the io$remove_fpd procedure could delete the principal IDs of all of the FPUs for this FPD, this is not done. It would only make sense to do so if the strategy was for the FPU remove routine to delete all of the primary object IDs for all of the channels assigned to the FPU. This is not done, however, for reasons discussed in Section 8.4.3.6.1.

Below is the format of the io$remove_fpd procedure:

```plaintext
PROCEDURE io$remove_fpd (  
  IN fpd: POINTER e$fpd_object_body;  
  IN access_mode: k$processor_mode;  
);

++
|
| Routine description:
| |
| Removes the fpd object from the system.
| Once removed, the fp can no longer be accessed by the system.
| |
| Arguments:
| |
| fpd reference pointer to fpd object body
| access_mode current mode
| |
| Return value:
| |
| node
| |
|--
```

8.4.2.4.2 io$delete_fpd

The Delete FPD (io$delete_fpd) I/O architecture procedure deletes a function processor definition and unloads the function processor code. Prior to unloading the function processor code, the function processor is called at its unload procedure via an address in the FPD. Note that, when the io$delete_fpd procedure is called, the pointer count on the FPD object is zero. This means there can be no outstanding requests to the function processor, all FPUs must already be deleted, and so on.

If the function processor still has any system threads, it must insure that they have exited before returning from the unload procedure. (Most function processors create threads to service function processor units and delete them in the respective FPU delete routine.)
The format of the `io$delete_fpd` procedure is as follows:

```
PROCEDURE io$delete_fpd (  
    IN fpd: POINTER e$fpd_object_body  
);
```

```
++
!
! Routine description:
!
! Prepares a fpd object for deletion by the object architecture.
! This routine calls the function processor Unload_FPD procedure.
!
! Arguments:
!
! fpd reference pointer to fpd object body
!
! Return value:
!
! none
!
!--
```

The `io$delete_fpd` procedure does the following:

```
BEGIN io$delete_fpd
    Get fp unload procedure address from fpd object
    Call the unload procedure
END io$delete_fpd
```

### 8.4.2.5 Unloading FPD

Before an FPD object can be deleted, the image of the function processor is deactivated. The FP unload procedure performs any necessary cleanup steps before the image is deactivated.

#### 8.4.2.5.1 e$unload_fpd

The format of the Unload FPD (e$unload_fpd) service is as follows:

```
PROCEDURE e$unload_fpd (  
    IN fpd_id: e$object_id;  
);
```

```
++
!
! Routine description:
!
! Calls the fp unload_fpd procedure to unload a function processor.
!
! Arguments:
!
! fpd_id fpd object id
!
! Return value:
!
! none
!
!--
```
The `e$unload_fpd` service routine does the following:

```
BEGIN e$unload_fpd

  Get reference pointer to FPD object body
  Set fpd_loaded variable in FPD object body to FALSE
  Get address of unload_fpd from FPD object body
  Call FP unload_fpd procedure

END e$unload_fpd
```

### 8.4.2.5.2 Function Processor's Unload FPD Procedure

The unload FPD procedure resides in the function processor's image. The purpose of this procedure is to prepare the function processor's image to be unloaded from the system. Before the function processor can be unloaded, all outstanding I/O requests must be canceled, and FPU objects and channel objects to the FPU deleted. Any data structures that have been allocated by the function processor must also be deleted. In addition, reference pointers to objects and object IDs must be disposed of properly. When the unload procedure returns, it is assumed that the function processor can be safely removed from the system.

The executive type definition for the unload FPD procedure is as follows:

```
TYPE
  e$unload_fpd_procedure:
    PROCEDURE () ;

PROCEDURE unload_fpd (
  ) OF TYPE e$unload_fpd_procedure;
```

```
!++
!
! Routine description:
!
! Unload a function processor
!
! Arguments:
!
!
! Return value:
!
! none
!
!--
```

### 8.4.3 FPU Functional Interface and Design

This section describes the details of the FPU object body structure, as well as the functional interface and design of each FPU object service routine.
### Table 8-9: FPU Object Services and I/O Architecture Procedures

<table>
<thead>
<tr>
<th>Service</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>exec$create_fpu</td>
<td>Create FPU object</td>
</tr>
<tr>
<td>eBind_fpus</td>
<td>Bind FPU objects together</td>
</tr>
<tr>
<td>e$Unbind_fpus</td>
<td>Unbind previously bound FPU objects</td>
</tr>
<tr>
<td>exec$set_fpu_information</td>
<td>Get or set information for FPU object</td>
</tr>
<tr>
<td>e$Register_ast_state_change</td>
<td>Registers a thread on an FPU</td>
</tr>
<tr>
<td>e$Deregister_ast_state_change</td>
<td>De-registers a thread on an FPU</td>
</tr>
<tr>
<td>e$change_fpu_state</td>
<td>Changes the state of an FPU object</td>
</tr>
<tr>
<td>io$State_change_est_cleanup</td>
<td>Cleans up the state change AST's data structure</td>
</tr>
<tr>
<td>e$verify_io_access</td>
<td>Verifies access to an I/O object</td>
</tr>
<tr>
<td>e$set_fpu_irp_size</td>
<td>Sets an FPU's IRP size</td>
</tr>
<tr>
<td>e$set_fpu_information</td>
<td>Executive form of exec$set_fpu_information</td>
</tr>
<tr>
<td>io$remove_fpu</td>
<td>Make FPU object unavailable to further I/O requests</td>
</tr>
<tr>
<td>io$delete_fpu</td>
<td>Prepare an FPU object for deletion</td>
</tr>
</tbody>
</table>

### 8.4.3.1 FPU Object Structure Definition

A function processor unit (FPU) is an object that represents a particular system resource, such as a volume or a device unit. When a channel object is created, it must specify the FPU object it is to be linked to.

An FPU always contains a function-processor-specific data structure that represents the specific type of unit or volume supported by the function processor.

#### Linkages

An FPU can be created in any object container.

Each FPU contains a function-processor-specific data structure for the unit it represents. An FPU also contains a pointer to the FPD associated with the function processor, as well as a pointer to an upper-level FPU, if this FPU is bound.

#### Allocation

FPUs are created by the exec$create_fpu and deleted by the exec$delete_object_id service.

#### Content

The FPU contains the following fields:

- kernel dispatcher event object, so an FPU can be waited on
- Mutex for synchronizing access to the FPU header
- Interface class code
- Maximum parameter area required for a request to this unit
- FPU state (see Section 8.3.2.3.1)
- Bound field (this field points to the upper FPU, if this FPU is bound)
- Bound FPUs (this list contains pointers to all FPUs that have been bound to this FPU)
- A set of disallowed access types, if the FPU is bound
- FPU sequence number (see Section 8.3.2.3.2)
- Number of channels connected to this FPU
• Cumulative total of all of the FP parameter records

FPU Object Body Structure
The FPU object body must be used by a function processor as the first field in the definition of its FPU object structure.
The structure of the body of the FPU object is described below.

```c
e$fpu_object_body: RECORD
    event: k$dispatcher_object(event);
    mutex: k$dispatcher_object(mutex);
    fpd: POINTER e$fpd_object_body;
    interface_class: e$sio_interface_class;
    state: e$fpu_state;
    bound: POINTER anytype;
    bound_fpus: e$linked_list;
    disallowed_access: SET [e$access_type];
    ast_state_change_listhead: POINTER e$sast_state_change_record;
    sequence_number: longword;
    channel_count: e$unsigned;
    cumulative_fp_param_record_size: e$unsigned;
END RECORD;
```

8.4.3.2 Creating FPU Object
An FPU object is created by calling the FPU create service (exec$create_fpu). This service is synchronous, so it only returns to the user when the request is complete. Error conditions are indicated by a status return.
The FPU may be deleted by the exec$delete_object_id service, which will result in a call to a Remove FPU procedure in the function processor. When the FPU object pointer count reaches zero, the function processor’s Delete FPU procedure is called.

8.4.3.2.1 exec$create_fpu
The Create FPU (exec$create_fpu) service creates an FPU and returns an object ID for it. Potential callers of this service include system management software, when creating device objects, and the MOUNT command, when mounting a disk or magnetic tape.
The function processor descriptor ID input parameter identifies the function processor for which the unit is to be created. Additional parameters indicate the object container the FPU is to be created in and, optionally, the FPU’s name and ACL value. These parameters are passed by the object_parameters record specified by the exec$create_fpu service routine.
The function processor may decide, based on the absence of an object name on input and the interpretation of item parameters, to determine the object name itself. For example, on a MOUNT operation, the function processor may derive the object name from the volume label on the media. In any case, the final name of the object is returned in the result_name parameter.
All parameters specific to the FPU being created are passed in via a special record defined by the function processor. The parameters will vary considerably, depending on what function processor is being used. For a MOUNT operation, the parameters might be a channel to the device to be mounted, volume label, mount options, and so on. For a device unit, they might be a description of the location of the controller, device unit number, and so on.
The create parameter record will have a fixed header containing an item list pointer similar to the I/O parameter record discussed later.
The exec$create_fpu call completes synchronously. The format of the call is as follows:

```
PROCEDURE exec$create_fpu (  
    OUT fpu_id: exec$object_id;  
    IN  fpu_object_parameters: exec$object_parameters;  
    IN  fpd_id: exec$object_id;  
    IN  fpu_parameters: POINTER anytype = DEFAULT CONFORM;  
) RETURNS STATUS;  
EXTERNAL;
```

++

Routine description:
++
I/O system service routine to create a function processor unit object.
++
Arguments:
++
  fpu_id          object id of created fpu object
  fpu_object_parameters parameters used by the object architecture when creating an object
  fpd_id          object id of fpd
  fpu_parameters  parameters used to initialize fpu object
++
Return value:
++
TBD
++

To create an FPU, the executive first attempts to verify the object ID specified for the FPD in the call. If the specified FPD is found, an FPU object is allocated (according to the size in the FPD) and initialized. A pointer is initialized to the FPD for the function processor, and the function-processor-specific data structure residing in the FPU object body is cleared. The FPU is not yet placed in the destination object container. Finally, the function processor is called at its Initialize FPU procedure, with pointers to the newly created FPU and the I/O parameter record specified in the call.

If the function processor encounters an error, then it must raise a condition. In this case, it must do whatever cleanup it requires as it unwinds to the caller. Upon receiving this signal, the exec$create_fpu service deletes the FPU and raises a condition to its caller.

Since FPU objects can be waited on, the exec$create_fpu service builds a kernel dispatcher event object into the FPU at creation. The additional storage for this object is calculated by exec$create_fpu. The purpose of the kernel dispatcher event object is to allow threads to wait on an FPU. The event object is initially cleared, and is set by exec$change_fpu_state whenever the FPU changes to a state other than TRANSITION.

If the function processor successfully returns from its Initialize FPU procedure, the common object service routine attempts to place the initialized FPU in the designated object container. If the insertion of the FPU into the object container fails, then the function processor's Delete FPU procedure is called to clean up and delete its structures. Upon return, the object service routine deletes the FPU object, and returns an appropriate error to the user.

If the FPU was successfully inserted into the object container, the object service routine returns successfully to the user.
The `e$create_fpu` service routine does the following:

BEGIN e$create_fpu

Probes Parameters
verify fpd object id
Get pointer to FPD object body
get size of FPU from FPD object body
add size of an kernel dispatcher event object to FPU object size
create FPU object (call e$create_and_initialize_object)
IF FPU object creation failed THEN
    set status - FPU object creation failure
ELSE
    Initialize kernel event object in FPU header
    Initialize FPU mutex
    Clear kernel dispatcher event object in FPU object
    Get address of initialize_fpu procedure from FPD object body
    Call initialize_fpu(fpuname, parameters, FPU_name)
    Insert object into container
    set status - success
ENDIF
END e$create_fpu

8.4.3.2.2 Function Processor’s Initialize FPU Procedure

The Initialize FPU procedure in a function processor is called by the `e$create_fpu` service. When the function processor is called, the FPU has been created and its fixed part initialized; the function-processor-specific part of the FPU (the size of which was taken from the FPD) has been cleared; and the FPU object has not yet been inserted in any object container. The function processor must do whatever specific validation and initialization it requires, and return or signal an error status.

If the function processor signals an error status, the Initialize FPU procedure raises a condition to signal the error status to its caller. If the function processor returns from its Initialize FPU procedure, the `e$create_fpu` service inserts the FPU object into the object container designated in the user call, then returns.

Note that the FPU object is not inserted in any object container until the Initialize FPU operation is successfully completed. This prevents the possibility of a channel being assigned to the FPU object while it is in the INTERMEDIATE state.

Each FPU contains the worst-case size of the IRP parameter area for any request to that FPU. For function processors that pass requests on to lower-level function processors, the IRP parameter area size from the lower-level FPU is added to the IRP parameter area size required for the current FPU. The total is then stored in the current FPU. In this way, the parameter size required for FPUs ripples up from the bottom.

Any attempt during the processing of Request I/O to allocate beyond the end of the available area is considered an inconsistency, which results in a bug check.

The function processor Initialize FPU procedure does the following:

- Validates user item parameters.
- Initializes the function-processor-specific data structure for the unit.

Some function processors may be finished at this point, not requiring the creation of threads. If the function processor requires threads, the Initialize FPU procedure can create a thread by calling `e$create_kernel_thread`.
The executive type definition for the Initialize FPU procedure is as follows:

```
TYPE
  e$initialize_fpu_procedure:
  PROCEDURE (
    IN fpu: POINTER e$fpu_object_body;
    IN fpu_parameters: POINTER anytype;
  );
```

The function processor defines its Initialize FPU procedure as follows:

```
PROCEDURE Initialize_fpu (
  fpu,
  fpu_parameters
) OF TYPE e$initialize_fpu_procedure;
```

```
++
!
Routine description:
!
This is the FP routine to initialize the body of the fpu object.
!
Arguments:
!
  fpu                      reference pointer to fpu object body
  fpu_parameters           parameters used to initialize fpu object
!

Return value:
!
  none
!
--
```

The function processor may itself decide how many system threads it requires, or it may be instructed how many to create via an entry in the `fpu_parameters` record. Normally, the system threads go directly into an idle state, waiting on a request queue in the FPU for IRPs to process. The use of system threads is discussed in Section 8.3.5.2.

If the function processor is a DFP for a direct-connected device, it needs to create a system thread for the unit. However, it first must do the following:

- If a thread is being created for the first unit on a multi-unit controller, a mutex, or some data structure containing a mutex, may have to be created to synchronize access to the controller by function processor threads.

- If a thread is being created for the first (or only) unit on a controller, the function processor normally has to connect an interrupt procedure to the interrupt vector for the controller, specifying a parameter (normally a data structure address) that is to be supplied as input to each invocation of the interrupt procedure.

A system thread is then created for the unit. The thread may start out by initializing the controller and/or device, noting the success/failure of initialization in the FPU. It then sets an event pointed to by the FPU to indicate when the device initialization has completed, and enters its idle state to wait on its request queue.

The function processor Initialize FPU procedure synchronizes on the event object in order to incorporate the status of the initialization into its final status. An error status from Initialize FPU causes a condition to be raised.

The use of system threads for driving physical devices is discussed in Section 8.3.5.3.
8.4.3.3 Binding FPU Objects

When an FPU is configured, there are services (such as MOUNT) that can be used to build successive virtual layers of FPUs on top of an existing FPU. When mounting a device, it is sometimes necessary to join one or more lower-level FPUs to a single upper-level FPU. This process is called binding FPU objects.

When the binding process completes, the system thread of the upper-level function processor that initiated the binding operation is referred to as the binder, and the lower-level FPU(s) are now bound to the FPU of the binder. Lower-level FPUs may only be bound if, and only if, there are no outstanding channels connected to them. However, if there are channels connected to any of the lower-level FPUs involved in the binding process, then the bind operation will fail. New lower-level FPU(s) can be added to an existing set of bound FPU(s), if there are no outstanding channels connected to them.

The e$bind_fpus service performs the binding operation, and guarantees that all lower-level FPUs specified in the call are bound to a binder’s FPU object when the call completes. If the service is unable to bind any one of the lower-level FPU objects specified in the call, then the binding operation fails. Also, at the time of binding, the binder may specify a set of access types that must not be granted to threads other than the binder. This set of access types is referred to as: disallowed_access_types. The set of disallowed_access_types is stored in each of the lower-level FPU objects, if the binding operation is successful.

If no access types are specified in the call, then the set is cleared. The FPU’s disallowed_access_types set is used by the e$verify_io_access service. The e$verify_io_access service only grants disallowed access types to the binder; if a disallowed access type is desired by a thread other than the binder, the access is denied and the service fails.

When the e$bind_fpus service returns after a failure, none of the lower-level FPUs are bound. The e$unbind_fpus service allows FPUs that were bound to be unbound individually, or all at the same time. For example, Step 1 in Figure 8–13 shows 3 FPU objects: FPUz, FPU1, and FPU2. A system thread can bind FPUz to FPU1 and FPU2 by calling the e$bind_fpus service. Step 2 in Figure 8–13 illustrates the relationship between the 3 FPUs after the e$bind_fpus service completes. For more detailed information on connecting channels to bound FPU objects, see Section 8.4.3.3.1.
Figure 8–13: Binding FPUs Objects

1. Determine which FPUs to bind

FPU₂ can bind FPU₁ and FPU₂ to itself, if there are no outstanding channel connections to either FPU₁ or FPU₂. The binding process is performed by the $\text{e$bind_fpus}$ service.

2. Bind FPU objects

The $\text{e$bind_fpus}$ service binds FPU₁ and FPU₂ to FPU₂ if and only if there are no outstanding channels connected to either FPU₁ or FPU₂.

8.4.3.3.1 Connecting Channels to a Bound FPU Object

Channels may be created to an FPU that is bound from either the function processor thread initiating the bind operation (the binder) or from other threads (assuming Mica security grants access to the FPU object). This section discusses the differences between an I/O request issued to a bound FPU from a binder, and an I/O request issued from another thread.
For example, step 1 in Figure 8–14 shows two FPU objects: $FPU_1$ and $FPU_2$. $FPU_1$ is bound to $FPU_2$. Function processor $Z$ has a system thread that creates channel $A$ to $FPU_1$ for $FPU_2$. Another thread creates channel $B$ to $FPU_1$. Even though there are two channels connected to $FPU_1$, I/O requests cannot be issued to $FPU_1$ unless the channels are accessed. For more detailed information regarding channel access, see Section 8.3.9.2.2.

Step 2 in Figure 8–14 illustrates an I/O request for FPU access issued by function processor $Z$'s thread on channel $A$. Before the access I/O request is issued, an I/O parameter record must first be built to store the parameters for the FPU access request. The parameters for the FPU access request are the $from\_binder$ field and a set of desired FPU access types. The $from\_binder$ field is set if this request comes from the binder; otherwise, it must be cleared. Even though this field is visible from user mode, it is only valid when set by a kernel-mode thread. If the $from\_binder$ field is set by a user-mode thread, then the I/O request is immediately rejected by the I/O architecture. Only kernel-mode threads are allowed to set the $from\_binder$ field in the parameter record for an FPU access I/O request and, therefore, are trusted to set this field only if they are the binder, or acting on behalf of the binder.

When a function processor processes an $io\_c\_fpu\_access$ function code, it makes a call to the $e\_verify\_io\_access$ service. The $e\_verify\_io\_access$ service expects a channel pointer, a set of $desired\_access$ types, and a $from\_binder$ parameter. If the accesses can be granted, then the set of $desired\_access$ types are stored in the channel, and $e\_verify\_io\_access$ succeeds. If the accesses cannot be granted, then the set of $allowed\_access$ types in the channel object remains unchanged.
Figure 8–14: Connecting Channels to a Bound FPU

1. Channel A and channel B are created to the bound FPU.

2. Issue FPU access function code to channel A

3. Issue FPU access function code to channel B

Note: If a user-mode thread tries to impersonate the binder, the request will be rejected by the I/O architecture.

Step 3 in Figure 8–14 illustrates a channel access I/O request issued on channel B. Since this request is not from a binder, the from_binder field in the I/O parameter record must be cleared. As stated before, only a kernel-mode thread can set the from_binder field. When the function processor processes the io$c_fpu_access function code, it must call $verify_io_access to determine if access can be granted. When the $verify_io_access service executes, it checks to see if the FPU being accessed has been bound. If the FPU has been bound, then the from_binder field is used to determine if the
request came from the binder. In this example, the request did not come from the binder, so the e$verify_io_access service must still determine if access can be granted.

If the following steps succeed, then the requested access is granted:

1. The set of disallowed access types stored in the FPU is retrieved and compared with the set of desired access types. If an access type is specified in both the desired access set and in the disallowed access set, then the access fails.

2. If step 1 succeeds, then the desired access types are checked by Mica security. If security denies the access types, the FPU access will fail; otherwise it is granted.

Only the function codes belonging to access types stored in the channel's granted access types set are accepted by exec$request_io. All other function codes are rejected.

If a thread, when issuing an FPU access I/O request, holds the identifiers necessary for granting a particular access type, the access type is not granted if all of the following are true:

- The thread is accessing a bound FPU
- The thread is not the binder
- The desired access type is disallowed for threads other than the binder

If a bound FPU is deleted, and the lower-level FPU(s) are still bound, then the I/O architecture unbinds the lower-level FPU(s) and clears the set of disallowed access types in the FPU object before the FPU object is deleted. This is done by the I/O architecture procedure: io$delete_fpu.

8.4.3.3.2 e$bind_fpsus

The format of e$bind_fpsus is as follows:

```
PROCEDURE e$bind_fpsus (   
    IN upper_level_fpu: e$object_id;   
    IN disallowed_access_types: SET [e$access_type];   
    IN lower_level_fpsus: e$object_id LIST;   
) RETURNS status;
```

|++|
|Routine description:|
|Binds one or more lower-level FPUs to a single upper-level FPU.|
|++|
|Arguments:|
|upper_level_fpu | object ID of the upper-level FPU |
|disallowed_access_types | a set of access types to be disallowed to |
|threads other than the binder. |
|lower_level_fpsus | a list of lower-level FPU object IDs |
|++|
|Return value:|
|Successful or Failed. |
|--
8.4.3.3  e$unbind_fpus

The e$unbind_fpus service unbinds the specified lower-level FPUs from the upper-level FPU object. When the last bound FPU is unbound, then the bound_fpu_list is deallocated. Only FPU objects that were bound together with the upper-level FPU can be unbound. In other words, it is not allowable to unbind FPU(s) that are not bound to the upper-level FPU specified in the call.

The format is as follows:

```plaintext
PROCEDURE e$unbind_fpus (  
    IN upper_level_fpu: e$object_id;  
    IN lower_level_fpus: e$object_id LIST;  
) RETURNS status;
```

Routine description:

Unbinds lower-level FPU(s) from an upper-level FPU.

Arguments:

upper_level_fpu  object ID of the upper-level FPU
lower_level_fpus  a list of lower-level FPU object IDs

Return value:

Successful or failed

8.4.3.4  Getting FPU Information

The exec$get_fpu_information service return select information about a target FPU.

8.4.3.4.1  exec$get_fpu_information

The Get FPU Information (exec$get_fpu_information) service is normally used for informational or performance monitoring purposes. A user program normally requests information about an FPU by issuing exec$request_io with the Get FPU Information function code after already creating a channel to the FPU.

The item list used in this call is identical with the list used by the Get FPU Information function code. However, if any fields are specified, which are not directly in the FPU and require asynchronous operation to retrieve the information, then an error is returned from exec$get_fpu_information.

The exec$get_fpu_information call completes synchronously. The format of the call is as follows:

```plaintext
PROCEDURE exec$get_fpu_information (  
    IN fpu_id: exec$object_id;  
    IN fpu_items: POINTER exec$item_list = DEFAULT;  
) RETURNS STATUS;  
EXTERNAL;
```
The `e$get_fpu_information` service routine does the following:

```c
BEGIN e$get_fpu_information
    Probe Parameters
    IF fpu object not valid THEN
        set status - Invalid FPU object
    ELSE
        Get pointer to fpu object body from channel object
        Get pointer to fpd object body from fpu object body
        Get address of the get_fpu_information procedure from
        FPD object body
        call get_fpu_information(fpu_pointer, items)
        set status - success
    ENDIF
END e$get_fpu_information
```

### 8.4.3.4.2 Function Processor's Get FPU Information Procedure

The purpose of the Get FPU Information procedure is to get the desired information from the FPU object, then return. The Get FPU Information procedure is defined in the image of the function processor and is not a part of the I/O architecture code. This procedure is dispatched from the FPD object when the `exec$get_fpu_information` service is executed. The type of information that can be requested is dependent upon each function processor. This call completes synchronously.

The executive type definition for the Get FPU Information procedure:

```c
TYPE
    e$get_fpu_information_proc:
    PROCEDURE ( 
        IN fpu: POINTER e$fpu_object_body;
        IN fpu_items: POINTER e$items_list = DEFAULT;
    );
```

The function processor defines its Get FPU Information procedure as follows:

```c
PROCEDURE get_fpu_information ( 
    fpu,
    fpu_items
) OF TYPE e$get_fpu_information_proc;
```
8.4.3.5 Misc. FPU Service Routines

8.4.3.5.1 e$register_ast_state_change

The e$register_ast_state_change executive service registers a thread with an FPU so that, if the FPU changes state, a state change AST is queued to the thread. This service allows a thread to register with multiple FPUs and, therefore, receive a state change AST from each FPU when a change of state occurs. If a thread registers more than once with the same FPU, only one AST block is queued to the thread. This service is used by the function processor when processing the io$c_enable_state_change_ast function code.

The format of e$register_ast_state_change is as follows:

```
PROCEDURE e$register_ast_state_change (  
    IN FPU: e$object_id;  
) INLINE;  
++  
|  
|  Routine description:  
|  
|  This routine registers a thread with an FPU object for  
|  receiving state change AST.  
|  
|  Arguments:  
|  
|  fpu        fpu object ID  
|  
|  Return value:  
|  
|  none  
|  
+--
```

The e$register_ast_state_change service executes as follows:

1. Allocates and initializes a state change data structure containing an AST block. This initialization process includes getting a pointer to the current thread and the previous mode from the thread's TCB.
2. Acquires the FPU head mutex
3. Inserts the AST state change record into the listhead
4. Releases mutex

8.4.3.5.2 e$s deregister_ast_state_change

The e$s deregister_ast_state_change service is used to stop a thread from receiving a state change AST. This is accomplished by canceling a thread's registration with an FPU. This service is used by the function processor when processing the io$s disable_state_change_ast function code.

The format for e$s deregister_ast_state_change is as follows:

```c
PROCEDURE e$s deregister_ast_state_change(
    IN fpu: POINTER e$s fpu object body;
) INLINE;
```

`++`  
Routine description:  
This service de-registers this thread from an FPU, so that no more state change ASTs can be received.  

Arguments:  
- fpu reference pointer to fpu object  

Return value:  

`--`

The process of de-registering a thread is as follows:
1. Locates the FPU object body fixed header (using the FPU ID)
2. Gets object ID for this thread
3. Acquires the FPU header mutex
4. Searches list for state change record for this thread
5. If an AST state change record is found for this thread, then the thread determines whether or not the record is in use by checking the state of the ast_in_queue flag
   a. If the AST state change record is in use, set the deallocate flag in the AST state change record
   b. If the AST state change record is not in use, deallocate it
6. Releases mutex.
8.4.3.5.3  e$change_fpu_state

The e$change_fpu_state service is used by a function processor to change the state of its FPU(s). This service changes the state of an FPU object according to the FPU state transition diagram presented earlier. If the new state specified in the call to e$change_fpu_state does not follow the state transition diagram, then an error condition occurs and the state of the FPU does not change.

This service executes as follows:

1. Acquires the FPU header mutex
2. Updates the sequence number in the FPU object
3. If the current FPU state is io$c_fpu_state_transition, then it signals the FPU’s kernel dispatcher event object. If the new FPU state is io$c_fpu_state_transition, then it clears the FPU’s kernel dispatcher event object.
4. For each AST state change record in the list, the service does the following:
   a. If the ast_in_queue flag is set, a state change AST has already been queued for this thread. Therefore, nothing is done.
   b. Checks the ast_in_queue flag.
   c. If the ast_in_queue flag is not set, then set it and queue a state change AST to the appropriate thread. An I/O piggyback AST, using a reference pointer to the FPU object as a parameter, is attached to the user AST. This piggyback AST executes the io$state_change_cleanup I/O architecture procedure.

5. Releases mutex.

The format for e$change_fpu_state is described below:

```
PROCEDURE e$change_fpu_state (  
  IN fpu: POINTER e$fpu_object_body;  
  IN new_state: e$fpu_state;  
  IN sequence_number: longword OPTIONAL;  
) RETURNS integer;  
```
8.4.3.5.4 \( \text{io\$state\_change\_ast\_cleanup} \)

The \( \text{io\$state\_change\_cleanup} \) function is invoked when the piggyback AST from a previously queued state change AST is executed. The purpose of this function is to clear the \text{ast\_in\_queue} flag in the AST state change record so that the record can be used again for queuing another state change AST. This function also deallocates the AST state change record, if it was marked for deallocation.

The format of \( \text{io\$state\_change\_cleanup} \) is as follows:

\[
\begin{align*}
\text{PROCEDURE } & \text{io\$state\_change\_cleanup} ( \\
& \text{IN } \text{ast\_state\_change\_record}: \text{POINTER e\$state\_change\_ast\_record}; \\
& )); \\
\end{align*}
\]

\[
\begin{align*}
\text{++} \\
\text{! Routine description:} \\
\text{!} \\
\text{! This procedure gets called by the state change piggyback AST} \\
\text{! to cleanup the state change AST block so that it can be reused.} \\
\text{! Arguments:} \\
\text{!} \\
\text{! ast\_state\_change\_record} \quad \text{pointer to the AST state change block} \\
\text{! that is to be cleaned up.} \\
\text{! Return value:} \\
\text{!} \\
\text{! none} \\
\text{!} \\
\end{align*}
\]

8.4.3.5.5 \( \text{e\$verify\_io\_access} \)

The \( \text{e\$verify\_io\_access} \) service is called by the function processor whenever an access function code is processed. This service verifies that the desired access can be granted. The service does this by making a call to Mica security to check if the set of desired access types is granted or denied. If Mica security grants the desired access, and the FPU pointed to by the channel is not bound, then the set of desired access types are stored in the channel. However, if the FPU pointed to by the channel is bound, and the \text{from\_binder} parameter is not set, then granted access types are checked against the disallowed access types stored in the FPU (see Section 8.4.3.3 for information on disallowed access types). If a match occurs, the service fails; otherwise it succeeds, and the set of desired access types are stored in the channel's \text{granted\_access\_types} set.

If a secondary object (an object other than an FPU object) is being accessed, this service requires a reference pointer to the secondary object, along with a pointer to its FCAT table. The \text{from\_binder} parameter is not used, since it has no meaning in this context.

This service verifies that the desired access types are granted to the secondary I/O object by calling Mica security with a reference pointer to the object and the set of desired access types. Note that the desired access types that are specified in the set must correspond to the access types for this object. Also note that the object's ACL is used by Mica security when performing the access checks. If Mica security grants the desired access types, then these access types are stored in the channel, along with the pointer to the object FCAT table.

The format of the \( \text{e\$verify\_io\_access} \) service is as follows:

\[
\begin{align*}
\text{PROCEDURE } & \text{e\$verify\_io\_access} ( \\
& \text{IN } \text{channel}: \text{POINTER e\$channel\_object\_body}; \\
& \text{IN } \text{desired\_fpu\_access}: \text{SET [e\$access\_type]}; \\
& \text{IN } \text{object\_type}: \text{POINTER anytype } = \text{DEFAULT CONFORM}; \\
& \text{IN } \text{object\_type}: \text{POINTER object\_type } = \text{DEFAULT}; \\
& \text{IN } \text{function\_code\_access\_table}: \text{POINTER e\$function\_codes\_access\_table } = \text{DEFAULT}; \\
& \text{IN } \text{from\_binder}: \text{Boolean}; \\
& )); \text{RETURNS boolean}; \\
\end{align*}
\]
Routine description:

This routine verifies that the desired access types can be granted to the I/O object. If an object is not specified, the FPU object pointed to by the channel is used. If the desired access types are granted then the allowed access set channel stores the granted access types.

Arguments:

- channel: reference pointer to channel object
- desired_fpu_access: A set of desired FPU access types
- object: object being accessed
- object_type: type of object
- function_code_access: table function code mapping table to use for this object.
- from_binder: Indicates if the request comes from the binder of a bound FPU object.

Return value:

- TRUE - if all access types can be granted.
- FALSE - if all access types cannot be granted

---

8.4.3.5.6 e$s$et_fpu_irp_size

This routine sets the value for the IRP size field in the header of the FPU object.

The format is as follows:

PROCEDURE e$s$et_fpu_irp_size (  
  IN fpu: POINTER e$fpu_object_body;  
  IN irp_size: e$unsigned;  
) RETURNS e$unsigned INLINE;

Routine description:

Set the value of irp_size within the header of the target FPU object.

Arguments:

- fpu: reference pointer to the target FPU object.
- irp_size: Maximum size needed for FP parameter record by this and all lower-level function processors.

Return value:

- type: e$unsigned
  - Value contained in the FPU's irp_size field.

8-64 I/O ARCHITECTURE
8.4.3.6 Removing an FPU

The I/O architecture defines two procedures for removing and deleting an FPU object. The \texttt{io$remove_fpu} procedure is called by the object architecture when the FPU object ID count goes to zero. The \texttt{io$delete_fpu} procedure is called by the object architecture when the FPU reference pointer count goes to zero. Each procedure calls a respective Remove FPU or Delete FPU procedure in the function processor.

8.4.3.6.1 \texttt{io$remove_fpu}

The function processor is called at its Remove FPU procedure to initiate any action it may require to accelerate the deletion. This might include dereferencing pointers, or even deleting channels to the FPU. Any threads that are still registered with this FPU are deregistered. This is done by issuing a synchronous request to \texttt{request io\_trusted} with the \texttt{io$c\_disable\_state\_change\_ast} function code.

No action is taken on existing channels. This is because it is not possible to delete channels at the job or process level, since the remove routine will, in general, be running in the wrong context. Even if it were possible, deleting all of the outstanding channels would have an unacceptable affect on the user of that channel. For example, deleting a channel belonging to a normally executing process, where the channel has a file accessed for write, could leave the file in an unacceptable state.

The format of the \texttt{io$remove_fpu} procedure is as follows:

```c
PROCEDURE io$remove_fpu (  
    IN fpu: POINTER e$fpu\_object\_body;  
    IN access_mode: k$processor\_mode;  
);  
```

The \texttt{io$remove_fpu} procedure does the following:

```c
BEGIN io$remove_fpu  
    Get pointer to fpd object from fpu object  
    Get fp remove_fpu procedure address from fpd object  
    IF threads have been registered for state change ast on this FPU THEN  
        deregistered the threads  
    Call the remove_fpu procedure  
END io$remove_fpu
```
8.4.3.6.2 Function Processor's Remove FPU Procedure

The function processor remove FPU procedure does any proactive processing to cause the FPU to be deleted. This could include requesting a dismount or unready action on the FPU, and deleting any channels to the FPU the function processor is responsible for. (Normally these actions are initiated via Request_IO services before the last FPU object ID is deleted.)

The executive type definition for the Remove FPU procedure is as follows:

```
TYPE
  e$remove_fpu_procedure:
    PROCEDURE ( 
      IN fpu: POINTER e$fpu_object_body;
      IN access_mode: k$processor_mode;
    );
```

The function processor defines its Remove FPU procedure as follows:

```
PROCEDURE remove_fpu ( 
  fpu,
  access_mode
) OF TYPE e$remove_fpu_procedure;
```

```
+++ 
| 
| Routine description: 
| 
| This is the function processor remove fpu procedure. This routine 
| delete any channels to the fpu that the FP is responsible for. 
| 
| Arguments:
| 
| fpu reference pointer to fpu object body 
| access_mode mode (user or kernel) 
| 
| Return value:
| 
| none 
| 
|--
```

8.4.3.6.3 io$delete_fpu

The FPU delete routine is called when the pointer count for the FPU is zero. This implies that there are no channels assigned, and no I/O requests can be outstanding. The delete routine retrieves the FPD pointer for the function processor from the body of the FPU object. It then calls the function processor's Delete FPU procedure, with the pointer to the FPU being deleted as a parameter.

The function processor performs whatever delete actions it requires. These actions include deleting data structures related to the FPU, and causing any system threads for the FPU to exit. The function processor then waits for these threads to exit. Upon return from the function processor FPU delete procedure, the FPD pointer in the FPU is released.

The format of the io$delete_fpu service is as follows:

```
PROCEDURE io$delete_fpu ( 
  IN fpu: POINTER e$fpu_object_body; 
); 
```
The `io$delete_fpu` routine does the following:

```plaintext
BEGIN io$delete_fpu
    Get pointer to fpd object from fpu object
    Get fp delete_fpu procedure address from fpd object
    Call the delete_fpu procedure
END io$delete_fpu
```

### 8.4.3.6.4 Function Processor's Delete FPU Procedure

The Delete FPU procedure does the reverse of the Create FPU procedure. Prior to calling the `io$delete_fpu` procedure, all channels to this unit have been deleted. This frees the function processor from having to synchronize the Delete FPU procedure with the requests to the unit.

Examples of some of the operations that a function processor may perform in its delete FPU routine are:

- Perform shutdown/unload operations for the unit.
- For the last unit on a controller of a DFP, disconnect the interrupt procedure, if the device is direct-connected.
- Signal any threads for the unit to exit, and wait for them to exit.
- Delete all data structures and objects that were created.

The executive type definition for the Delete FPU procedure is as follows:

```plaintext
TYPE
    e$delete_fpu Procedure:
        PROCEDURE (fpu: POINTER e$fpu_object body;
)

The function processor defines its Delete FPU procedure as follows:

```plaintext
PROCEDURE delete_fpu (fpu
) OF TYPE e$delete_fpu Procedure;
```
Routine description:

This is the function processor delete fpu procedure.

Arguments:

fpu  reference pointer to fpu object body.

Return value:

none
8.4.4 Channel Functional Interface and Design

This section describes the structure of the channel object and the service routines used to manipulate it. Listed below are the channel object services defined by the I/O Architecture.

<table>
<thead>
<tr>
<th>Service</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>exec$create_channel</td>
<td>Create channel object to FPU</td>
</tr>
<tr>
<td>io$remove_channel</td>
<td>Inhibit further requests to this channel</td>
</tr>
<tr>
<td>io$delete_channel</td>
<td>Prepare the channel for deletion</td>
</tr>
<tr>
<td>exec$get_channel_information</td>
<td>Get information on channel</td>
</tr>
<tr>
<td>e$get_channel_information</td>
<td>Executive form of exec$get_channel_information</td>
</tr>
<tr>
<td>exec$request_io</td>
<td>Initiate an I/O request</td>
</tr>
<tr>
<td>e$request_io</td>
<td>Executive form of exec$request_io</td>
</tr>
<tr>
<td>e$request_io_trusted</td>
<td>A trusted executive form of exec$request_io; less parameter checking is done</td>
</tr>
<tr>
<td>ioInitialize_io_parameters</td>
<td>Build IRP and initialize it via parameters passed by exec$request_io_trusted</td>
</tr>
<tr>
<td>e$execute_io</td>
<td>Perform requested I/O operation</td>
</tr>
<tr>
<td>e$synchronous_io_call</td>
<td>Special internal service used for special interfaces between function processors</td>
</tr>
<tr>
<td>e$set_callback_table</td>
<td>Sets the callback table pointer in channel object</td>
</tr>
<tr>
<td>e$remove_callback_table</td>
<td>Removes callback table pointer from channel object</td>
</tr>
<tr>
<td>e$set_channel_to_accessed</td>
<td>Marks the channel as being accessed</td>
</tr>
<tr>
<td>e$clear_channel_access</td>
<td>Marks the channel as not being accessed</td>
</tr>
</tbody>
</table>

Of the channel object services listed, the exec$create_channel service completes on its own without a call to the function processor. When a channel is deleted via a call to exec$delete_object_id, a Cancel is first issued on the channel, and a deaccess call is issued if the channel is accessed. Cancel calls may result in calls to a function processor, but they are not dispatched to the function processor via the normal FPD mechanism. (The dispatching mechanism for Cancel will be discussed later.)

All normal I/O functions in Mica are performed via the exec$request_io service. This service is very similar to sys$qio and sys$qiw in VMS, however exec$request_io passes I/O parameters by means of an FP parameter record, as discussed in Section 8.4.1.1.2.

8.4.4.1 Channel Object Structure Definition

The Channel object describes a logical path from an object container to an FPU for performing I/O. It is a variable-length structure with a fixed header and a function-processor-specific area. The length of the function-processor-specific area in the channel object is stored in the FPD for the respective function processor.

Linkages

Channel objects may potentially be found in any object container in the system.

The channel object points to its respective FPU.

The channel object also has a listhead of IRPs for outstanding requests on this channel.

If a Deaccess operation is performed on a channel that has outstanding I/O requests, a pointer to the IRP for the Deaccess operation will be stored in the channel object while the outstanding I/O requests run down.
Allocation
The channel object is allocated by the exec$create_channel service, and is deallocated by a call to exec$delete_object_id after everything has run down and has been deaccessed on the channel.

Content
The channel object contains:

- A count of outstanding I/O requests issued on the channel.
- An accessed flag, which is initialized to FALSE and thereafter only modified by the function processor. If the accessed flag is TRUE when the channel is deleted, a deaccess function is issued on the channel from the channel object remove routine.
- A function-processor-specific variable-length area. The size of this area is recorded in the FPD. This area is initially cleared by the channel object create routine when the channel is created.
- The FPU sequence number from the last time the channel was synchronized with the FPU state. See Section 8.3.2.3.2.
- A mutex for synchronizing access to the channel header.
- A field that points to the callback table in this channel object.
- A set of granted access types.
- A pointer to the function processor’s function code access table.
- A pointer to an IRP containing a deaccess request.

Channel Object Body Structure
The channel object body must be used by a function processor as the first field in the definition of its channel object structure.

The body of the channel object is as follows:

```c
exec$channel_object_body: RECORD
  channel_mutex : k$dispatcher_object(mutex);
  outstanding_irp_list: exec$linked_list;
  channel_access: boolean;
  fpu-pointer: POINTER exec$fpu_object_body;
  channel_sequence_number: longword;
  channel_attribute: exec$channel_attribute;
  channel_callback_table: POINTER anytype;
  granted_access_types: SET [exec$access_type];
  function_code_access_table: POINTER exec$function_code_access_table;
  channel_deaccess_irp: exec$io_request_packet
END RECORD;
```

Channel Assignment Examples
The following three figures depict the data structures previously discussed (as well as some file system data structures) to show:

1. A channel assigned to a physical device unit
2. A channel assigned to an accessed file on a volume
3. A channel assigned to an accessed file on a volume set

8–70 I/O ARCHITECTURE
Figure 8-15: Channel Assigned to Device Unit with DFP

IRP → CHANNEL OBJECT

I/O request is passed to the FP via the channel

Assigned to Device FPU

Device's FPU OBJECT

Device's FPD OBJECT

Dispatched Table

pointers to FP entry points

DEVICE FUNCTION PROCESSOR

Figure 8-16: Channel Assigned to Volume with VFP

IRP → channel object

reference pointer

FPU RVT VCB

reference pointer

FPD object

pointers to entry points

Volume Function Processor

channel object

FPU object

Device Function Processor
8.4.4.2 Creating a Channel

The create channel services create a channel object, which is a logical I/O path to an FPU.

Note that the FPU for the channel is determined at the time the create channel service is processed, and that this is the only FPU that can be accessed directly on this channel. For example, when a channel is created for a file volume, it is impossible to access the underlying disk directly on this channel. The intent is to build a robust solution, which should, for example, help users who have expanded privileges to avoid having program errors clobber disks.
8.4.4.2.1 exec$create_channel

The Create Channel (exec$create_channel) service completes synchronously.

The exec$create_channel service creates a channel object. When exec$create_channel is called and
the object is created, a check is made as to whether the issuer can access the object. This check is
a side effect from the call to es$translate_id for the fpu_id parameter. If es$translate_id succeeds, it
returns to the channel object a pointer to the FPU object. The es$translate_id procedure is discussed
in Chapter 5, Object Architecture.

Note that Mica supports no implicit allocation when a channel is assigned. If an FPU requires
allocation, then the caller must allocate it before creating the channel.

The format of the exec$create_channel service is as follows:

PROCEDURE exec$create_channel (  
    OUT channel_id: exec$object_id;  
    IN channel_object_params: exec$object_parameters;  
    IN fpu_id: exec$object_id;  
) RETURNS STATUS;  
EXTERNAL;

+++  
! Routine description:  
!  
    I/O Architecture system service for creating a channel object to a fpu.  
!  
    Arguments:  
    
    channel_id   object id of created channel object  
    channel_object_params   object architecture create object parameters  
    fpu_id   id of fpu object to which channel is created  

! Return value:  
!  
! TBD  
!  
The es$create_channel service routine does the following:

BEGIN es$create_channel

Probe Parameters
IF fpu object not valid THEN
    set status - Invalid FPU object id
ELSE
    Get fpu pointer
    Get fpd object pointer from fpu object
    get channel size from fpd object
    calculate true channel size
    create a channel object via a
call to es$create_and_initialize_object
    IF channel object was created THEN
        store channel id in supplied variable
        Initialize channel object body
        Insert Channel object into Channel container
        Acquire FPU header mutex
        increment FPU's channel count
        Release FPU header mutex
        set status - Success
    ELSE
        set status - Object Creation error
    ENDIF
ENDIF

I/O ARCHITECTURE 8-73
END e$create_channel

Of the channel object calls listed, exec$create_channel is completed directly in the channel object services and does not result in a call to the function processor. When a channel is removed by a call to exec$delete_object_id, the e$cancel_io_by_channel service is called with a pointer to the channel object body as a parameter. The e$cancel_io_by_channel service may result in calls to a function processor, but they are not dispatched to the function processor via the normal FPD mechanism. (The dispatching mechanism for e$cancel_io_by_channel is discussed later.)

8.4.4.3 Removing and Deleting a Channel

Channels are removed and deleted by the I/O architecture procedures: io$remove_channel and io$delete_channel. The io$remove_channel routines is called when all of the channel's object ID has been deleted. The io$delete_channel routine is called when the channel's reference count reaches zero.

8.4.4.3.1 io$remove_channel

The io$remove_channel procedure first attempts to cancel any outstanding requests on the channel. If the accessed flag in the channel object is TRUE, this procedure calls the function processor with a deaccess call on the channel. This deaccess call is issued by an internal call to e$request_io_trusted (see Section 8.4.4.5.2), with a pointer to the channel object as input. The function processor must then take whatever action it requires for deaccess, such as deaccessing a file, closing a virtual circuit, or whatever.

The format of the io$remove_channel procedure is as follows:

```
PROCEDURE io$remove_channel (    
    IN channel: POINTER e$channel_object_body CONFORM; 
); 

+++ 
! Routine description: 
!   Removes a channel from the system. 
! Arguments: 
!   channel reference pointer to body of channel object to be removed 
! Return value: 
!   node 
!--
```

The io$remove_channel procedure does the following:

```
BEGIN io$remove_channel
    Cancel all outstanding request on this channel 
    (call e$cancel_io_by_channel)
    IF access flag in channel body is TRUE THEN 
      issue a synchronous request_io to the fpu with a 
      deaccess function code 
    ENDIF
END
END io_remove_channel
```
8.4.4.3.2  io$delete_channel

The io$delete_channel procedure simply releases its pointer to the FPU.

The format of the io$delete_channel procedure is as follows:

```
PROCEDURE io$delete_channel (  
    IN channel: POINTER e$channel_object_body CONFORM;  
);
```

The io$delete_channel procedure does the following:

```
BEGIN io$delete_channel  
  Acquire FPU's header mutex  
  decrement FPU's channel count  
  Release FPU header mutex  
  dereference fpu object  
END io$delete_channel
```

8.4.4.4  Get Channel Information

Information about a channel is requested by the Get Channel Information (exec$get_channel_information) service.

8.4.4.4.1  exec$get_channel_information

The exec$get_channel_information service returns selected information about the target channel. It returns only information about the channel itself.

The exec$get_channel_information service completes synchronously. The format is as follows:

```
PROCEDURE exec$get_channel_information (  
    IN channel_id: exec$object_id;  
    IN channel_items: POINTER exec$item_list = DEFAULT;  
) RETURNS STATUS;  
EXTERNAL;
```
8.4.4.5 Request I/O Description

The Request I/O (exec$request_io) service and its executive counterparts: e$request_io, e$request_io_trusted, io$initialize_io_parameters, e$execute_io, provide the mechanisms necessary to issue and propagate an I/O request.

8.4.4.5.1 exec$request_io

All normal I/O functions are performed through the exec$request_io system service. Details of this service are discussed in Section 8.3.3.

The format of the exec$request_io is as follows:

PROCEDURE exec$request_io (
    IN channel_id: exec$object_id;
    IN function_code: integer;
    BIND iosb: exec$iosb;
    IN completion_event: exec$object_id = DEFAULT;
    IN completion_ast: exec$sast_procedure = DEFAULT;
    IN ast_parameter: integer = DEFAULT;
    IN io_parameters: POINTER anytype = DEFAULT;
) RETURNS status;
EXTERNAL;

+++ 
++ 
++ Routine description: 
++ 
++ This is I/O Architecture system services interface for requesting I/O. 
++ 
++ Arguments: 
++ 
++ channel_id object id of channel to request io on 
++ function_code I/O request function code 
++ iosb I/O status block 
++ completion_event user event object to be signaled after I/O completion 
++ completion_ast ast procedure address to be called when the I/O completes. 
++ ast_parameter parameter for ast procedure 
++ io_parameters pointer to I/O parameter record 
++ 
++ Return value: 
++ TBD 
++ 
+++
The $request\_io$ service routine does the following:

```
BEGIN $request\_io

verify objects (channel, event)
Get a channel reference pointer
Determine access_mode
Get function code access table access pointer from channel
Retrieve function code access types from function code
access_table.
IF function code's access types are not in the channel's
granted access types set THEN
  return with error status
ELSE
  call $request\_io\_trusted$ with appropriate arguments

END $request\_io$
```

8.4.4.5.2 $request\_io\_trusted$

The Request I/O services build I/O requests into I/O request packets (IRPs), pass them to function processors, and implement I/O completion and I/O rundown/cancellation. Request I/O services are also responsible for making I/O requests asynchronous with an event object and AST notification, I/O status posting, and the final buffer copy for buffered I/O requests.

The Request I/O services provide a trusted ($request\_io\_trusted$) interface with less parameter checking. Internal calls generating new I/O requests call either $request\_io\_trusted$ or $request\_io$. There is no other way to generate a new IRP.

Note that executive routines and function processors must only call $request\_io\_trusted$. The $request\_io$ service is currently only intended to support user calls via the system service. The $request\_io\_trusted$ service has a similar interface to $request\_io$, except that it accepts pointers to the channel and event object data structures. It also accepts an originating access mode parameter (user or kernel). The format of the $request\_io$ service is as follows:

```
PROCEDURE $request\_io\_trusted$ (IN channel: POINTER $channel\_object\_body$;
IN function_code: integer;
BIND iosb: $iosb$;
IN completion_event: POINTER $event$ = DEFAULT;
IN kernel_event: POINTER $k\_dispatcher\_object(event) = DEFAULT$;
IN completion_ast: exec$ast\_procedure = DEFAULT$;
IN ast\_parameter: integer = DEFAULT$;
IN io\_parameters: POINTER anytype = DEFAULT$;
IN release\_channel: boolean = true$;
IN access\_mode: k\_processor\_mode$;
);
EXTERNAL;
```

/*
! Routine description:
!
This is a trusted interface for requesting I/O.
!
Arguments:
!
channel
function_code    reference pointer of channel to request io on
iosb
io\_status block
completion\_event reference pointer to user event object to be
signaled after I/O completion
kernel\_event reference pointer to kernel event object to
be signaled after I/O completion
completion\_ast ast procedure address to be called when the
I/O completes.
*/
Note the release_channel parameter and the fact that either a user or kernel event object may be specified.

A user event object will normally only be specified in calls originating from user mode with an event object specified. Also, in calls from user mode, the release_channel parameter will always be specified as TRUE. When such a call is completed, the channel reference pointer is released. The event object reference pointer is also released if one was specified.

Internal calls to e$request_io_trusted have a little more freedom. The caller may decide to increment the reference count of the channel object for a request and specify release_channel as TRUE, which causes the reference pointer to be decremented by the completion procedure. Alternately, the caller may decide that this is unnecessary and that it already has the channel count incremented, in which case it would specify release_channel as FALSE.

In addition, if the caller wishes to specify an event object, the caller may choose to either pass a pointer to a user event object, or to a kernel event object imbedded within another structure. If a user object is specified, the pointer to it will be released, which means the caller must make sure that it does a matching reference for this call. There is no release for kernel event objects.

The e$request_io_trusted service routine does the following:

BEGIN e$request_io_trusted

Get fpu object reference pointer from channel object
Save current AST state
Disable kernel-mode ASTs
call io$initialize_io_parameters(fpu, function, irp, parameters, access_mode)
initialize irp header
Acquire channel mutex
Link IRP to channel IRP Queue
Write channel pointer in irp header
Release channel mutex
Get pointer to thread object and store in irp
Link irp to thread IRP queue
Store thread reference pointer in irp
IF function code is EQUAL to deaccess and channel’s outstanding IRP list
  is not empty THEN
  IF channel’s IRP deaccess pointer NOT EQUAL to NIL THEN
    save IRP pointer in channel object
ELSE
Raise condition (2nd deaccess request)
ENDIF
ELSE
call e$execute_io(channel, function, irp, access_mode)
ENDIF
restore previous AST state
IF this is a synchronous request THEN
call e$synchronize_io_trusted (thread’s i/o event object, iosb)
ENDIF

END e$request_io_trusted

8-78 I/O ARCHITECTURE
An I/O architecture procedure and two executive services deal with I/O requests. They are discussed in detail in the following subsections.

- $ioinitialize_io_parameters$. This is an untrusted interface for requests originating from user mode, but some parameters are trusted for requests originating from kernel mode.
- $execute_io$. This is a trusted interface which accepts an I/O request as described by an IRP.
- $synchronous_io_call$. This is a trusted interface which accepts a parameter record as input and returns synchronously as a procedure.

The $request_io_trusted$ service is processed as follows:

1. The $request_io_trusted$ service begins by obtaining the reference pointer to the FPU object from the channel object and disabling kernel-mode ASTs. The function processor is called at its $initialize_io_parameters$ entry point via the $initialize_io_parameters$ procedure.

   Note that kernel mode ASTs are disabled just prior to calling $initialize_io_parameters$, and are enabled upon return from $execute_io$. This is done to help the function processor synchronize with completion procedures running in a special kernel mode AST in the same thread. This is only necessary when generating a new request for the current thread.

2. The $initialize_io_parameters$ service accepts an I/O parameter record and an originating access mode (kernel or user) as input. This service then transfers control directly to the function processor. The function processor allocates the IRP (via a call to $allocate_irp$), then possibly allocates storage for an HTL or buffers, if the request requires direct of buffered I/O, respectively. The $get_size_of_htl$ service is used to determine the size of an HTL for storage allocation. The $initialize_io_parameters$ service also allocates an FP parameter record in the IRP (via a call to $allocate_fp_parameter_record$), verifies the parameters in the I/O parameter record, and captures these parameters in an internal format for the newly created FP parameter record.

   The function processor normally minimizes or eliminates most parameter checks if the originating access mode is kernel.

   The function processor raises a condition if an error is detected during parameter verification.

   After the function processor has processed the parameters, control returns to $request_io_trusted$. The I/O is now ready for execution via the initialized IRP.

   Note that any errors prior to this point raise a condition to the caller of $request_io_trusted$, which, in the case of a user request, signal the caller of $request_io$. The $request_io_trusted$ service guarantees that if a valid event was specified, it will be signaled; but if an AST procedure was specified, it will not be called. The IOSB remains cleared.

   If no condition has been raised prior to this point, then $request_io_trusted$, as well as $request_io$ for a user call, will complete successfully after calling the $execute_io$ service described in the next step.

   The I/O at this point is said to be "successfully requested." An error during the execution of the request is returned in the IOSB. If an event was specified, it is signaled. If an AST procedure was specified, it will be called.

3. The IRP is passed to the function processor via a call to the $execute_io$ executive service. The function processor must now perform the specified I/O request in the manner described in Section 8.3.5.

   The function processor is expected to return from this call. If an exception is generated at this point, a bug check occurs.

   On return from the function processor, $request_io_trusted$ returns to its caller, if this was an asynchronous request. In the case of a user call via $request_io$, the return propagates to user mode.
During a synchronous I/O request, return from e$\text{request}_\text{io}_{\text{trusted}}$ is delayed by a call to e$\text{synchronize}_\text{io}_{\text{trusted}}$ with an internal thread-specific event. When the I/O completes, the IOBS is written and the internal thread-specific event signaled. At this point, the e$\text{synchronize}_{\text{io}_{\text{trusted}}}$ service returns to its caller.

One of the next two steps takes place later at an asynchronous point in time to complete the actual requested I/O represented by the IRP:

- The function processor always calls e$\text{complete}_\text{io}$ to complete the request and post the status to the user. Since this, in general, occurs asynchronously from the request, the bulk of the I/O completion processing occurs in a kernel-mode AST procedure for the issuing thread.

- Potentially, the request may be canceled before completion, via either the channel object or the thread object to which it is linked. However, this still results in the request being completed via e$\text{complete}_\text{io}$.

### 8.4.4.5.3 io$\text{initialize}_\text{io}_{\text{parameters}}$

The initialize I/O parameters (io$\text{initialize}_\text{io}_{\text{parameters}}$) procedure is used to allocate, verify, and initialize the IRP with the parameters issued by the I/O request.

The format of io$\text{initialize}_\text{io}_{\text{parameters}}$ is as follows:

```c
PROCEDURE io$\text{initialize}_\text{io}_{\text{parameters}}$

    OUT irp: POINTER e$\text{io}_{\text{request}}$ _packet;
    IN fpu: POINTER e$\text{fpu}_{\text{object}}$ _body;
    IN function_code: integer;
    IN parameters: POINTER anytype;
    IN access_mode: k$\text{processor}_{\text{mode}}$;
);
```

++++
!
! Routine description:
!
!   Calls the function processor initialize io parameters procedure and
!   initializes the irp.
!
! Arguments:
!
!   irp               pointer to allocated irp
!   fpu              reference pointer to fpu object body
!   function_code   I/O request function code
!   parameters      pointer to I/O parameter record
!   access_mode     requester processor mode
!
! Return value:
!
! none
!
--

The io$\text{initialize}_\text{io}_{\text{parameters}}$ procedure does the following:

BEGIN io$\text{initialize}_\text{io}_{\text{parameters}}$

    Get fpd object reference pointer from fpu object
    Get pointer to fpd object body from fpu object body
    Get address of the initialize_io_parameters procedure from
    FPD object body
    call initialize_io_parameters(function, irp, parameters, access_mode)
END e$\text{initialize}_\text{io}_{\text{parameters}}$

8-80 I/O ARCHITECTURE
The **$initialize_io_parameters** procedure is called by the I/O architecture from $request_io_trusted during the processing of a new user request via exec$request_io, or a new kernel-mode request via a direct call to $request_io or $request_io_trusted. The channel for the request is located in the IRP. Some checks may be reduced or eliminated if access_mode is kernel.

### 8.4.4.5.4 Function Processor's Initialize I/O Parameters Procedure

This procedure is called when the executive wants the function processor to verify the parameters of a new request and initialize the IRP with these parameters. The Initialize I/O Parameters procedure translates the parameters issued by the I/O request into an internal format and writes them as an FP parameter record into the IRP's free area.

The function processor has the responsibility to allocate the IRP. The I/O architecture has provided an executive service called $allocate_irp to allocate storage for an IRP. Before calling this service, the function processor should determine how much additional storage is required, based on the I/O request function code. For example, if a function processor receives a request to do direct IO, it should combine the size of the host transfer list (HTL) with the IRP size (taken from the FPU object) before calling $allocate_irp. After the IRP is allocated, the function processor can then reserve a portion of the IRP for the HTL. By doing this, a second call to the memory allocator can be eliminated. If an error occurs after the IRP has been allocated, the function processor can deallocate the IRP using the $deallocate_irp service.

The **$initialize_io_parameters** service is a jacket routine for this procedure. They both have the same interface.

For further details see Section 8.3.4. The executive type definition for the Initialize I/O Parameters procedure is as follows:

```plaintext
TYPE
  $initialize_io_parameters_proc:
    PROCEDURE (
      OUT irp: POINTER $io_request_packet;
      IN fpu: POINTER $fpu_object_body;
      IN function_code: integer;
      IN parameters: POINTER anytype CONFORM;
      IN access_mode: $processor_mode;
    );
```

The function processor defines its Initialize I/O Parameters procedure as follows:

```plaintext
PROCEDURE initialize_io_parameters (
  irp,
  fpu,
  function_code,
  parameters,
  access_mode,
) OF TYPE $init_io_parameters_proc;
```
8.4.4.5.5 \texttt{e$execute\_io} \\

The execute I/O (\texttt{e$execute\_io}) service is used to pass an IRP to a function processor for execution. This service can be issued either by \texttt{e$request\_io\_trusted}, or by a function processor to pass the request on to another function processor.

The format of the \texttt{e$execute\_io} service is as follows:

\begin{verbatim}
PROCEDURE e$execute_io (  
  IN channel: POINTER e$channel\_object\_body;  
  IN function_code: integer;  
  IN irp: POINTER e$io\_request\_packet;  
  IN access_mode: POINTER anytype;  
);  
EXTERNAL;
\end{verbatim}

The \texttt{e$execute\_io} service routine does the following:
BEGIN e$execute_io

Get pointer to fpu object body from channel object
Get pointer to fpd object body from fpu object body
Get address of the execute_io procedure from
FPD object body
call e$clear_cancel_procedure(irp) this will remove the
cancel procedure from the irp
call execute_io(function,irp,parameters,access_mode)

END e$execute_io

8.4.4.5.6 Function Processor’s Execute I/O Procedure

This procedure is called to pass an I/O request as represented by an IRP to a function processor for
execution.

The Execute I/O procedure may be called to initiate the execution of a new request from e$request_
io_trusted. In this case, the channel input parameter is the same as the channel stored in the IRP.

The Execute I/O procedure may also be called by a function processor to pass an existing request
on to another function processor via a call to e$execute_io. In this case, the channel parameter is
a channel to the FPU of the second function processor. This channel should have been created or
acquired by the first function processor. The channel input parameter in this case is not the same as
the channel in the IRP.

Note that function processors passing requests internally to other function processors via calls to
e$execute_io must have intimate knowledge of the interface to the function processor to which the
request is being passed. It is a general design rule in the system that a function processor may only
have such intimate knowledge of other function processors to which it directly passes requests; it may
not assume anything about lower-level function processors to which the request may be subsequently
passed.

The e$execute_io service is a jacket routine for this function processor procedure, and it has the same
interface.

For further details on the execution of I/O requests see Section 8.3.5.

The executive type definition for the Execute I/O procedure is as follows:

TYPE
    e$execute_io:
    PROCEDURE (  
        IN channel: e$channel_object_body;
        IN function_code: integer;
        IN irp: e$io_request_packet;
    );

The function processor defines its Execute I/O procedure as follows:

PROCEDURE execute_io (  
    channel,
    function_code,
    irp
);
8.4.4.5.7 e$synchronous_io_call

This is a trusted internal interface which has no corresponding user call. This service is used for special tailored interfaces between function processors and is part of the design of the function processors involved. Therefore, there is no further discussion of this service in this chapter.

The format of the e$synchronous_io_call service is as follows:

```
PROCEDURE e$synchronous_io_call (
   IN channel: POINTER e$channel_object_body;
   IN function_code: integer;
   IN io_call_parameters: POINTER anytype CONFORM;
);
EXTERNAL;
```

The e$synchronous_io_call service routine does the following:

```
BEGIN e$synchronous_io_call
   Get pointer to FPU object from channel object
   Get pointer to FPD object body from FPU object body
   Get address of synchronous_io_call from FPD object body
   Call synchronous_io_call(function, parameters)
END e$synchronous_io_call
```
8.4.4.5.8 Function Processor's Synchronous I/O Call Procedure

The Synchronous I/O Call procedure supports synchronous calls that perform a short function, such as buffer allocation. This procedure allows short functions to be initiated when it is desirable to avoid the cost of context switching and the creation of an additional IRP.

The format of the Synchronous I/O Call procedure is as follows:

PROCEDURE synchronous_io_call (  
  IN channel: POINTER e$channel_object_body;  
  IN function_code: integer;  
  IN io_call_parameters: POINTER anytype CONFORM;  
);

++
|
| Routine description:
|
|
| Arguments:
| channel reference pointer to channel object. This must be the channel for the fpu receiving the synchronous io call.
| function_code io function call
| io_call_parameters parameters for the io call
|
| Return value:
| none
|
|--

The e$synchronous_io_call service is a jacket routine for this function processor entry, and it has the same interface.

8.4.4.6 Misc. Channel Services

8.4.4.6.1 e$set_callback_table

The e$set_callback_table service stores a pointer to the callback table in the channel header and sets the channel access field.

The format of the e$set_callback_table service is as follows:

PROCEDURE e$set_callback_table (  
  IN channel: POINTER e$channel_object_body;  
  IN callback_table: POINTER anytype;  
 ) INLINE ;

++
|
| Routine description:
| Stores the pointer to the callback table in the channel header and sets the channel access flag.
|
| Arguments:
| channel reference pointer to the channel object
| callback_table pointer to the callback table
|
| Return value:
| none
8.4.4.6.2 e$remove_callback_table

The e$remove_callback_table service removes the pointer to the callback table from the channel header.

The format is as follows:

PROCEDURE e$remove_callback_table (  
    IN channel: POINTER e$channel_object_body;  
    IN callback_table: POINTER anytype;  
) INLINE;

+++  
|  
|  ! Routine description:
|  
|  ! Clears the callback table pointer field in the header of the channel object.
|  
|  ! Arguments:
|  
|  ! channel reference pointer to the channel object
|  
|  ! callback_table pointer to the callback table
|  
|  !
|  
|  ! Return value:
|  
|  !  none
|  
|---

8.4.4.6.3 e$set_channel_access

This routine sets the state of the channel access field.

The format is as follows:

PROCEDURE e$set_channel_access (  
    IN channel: POINTER e$channel_object_body;  
) INLINE;

+++  
|  
|  ! Routine description:
|  
|  ! Sets the channel access field in the header of the channel object.
|  
|  ! Arguments:
|  
|  ! channel reference pointer to the channel object
|  
|  ! Return value:
|  
|  !  none
|  
|---
8.4.4.6.4 e$clear_channel_access

This routine clears the channel access flag and the granted access types set in the channel.

The format is as follows:

```c
PROCEDURE e$clear_channel_access (  
    IN channel: POINTER e$channel_object_body;  
) INLINE;

++
|
| Routine description:
| |
| Clears the channel access flag and the granted
| access types set in the channel object.
| |
| Arguments:
| |
| channel    reference pointer to the channel object
| |
| Return value:
| |
| none
| |
|--
```

8.4.4.6.5 e$synch_channel_with_fpu

The e$synch_channel_with_fpu service provides a mechanism for synchronizing the sequence number in the channel object with that of the FPU object. Consider a situation in which an FPU object changes state and its sequence number is incremented via the e$change_fpu_state, but the sequence number in the channel object remains unchanged. In such a situation, the channel can be synchronized with the FPU by calling the e$synch_channel_with_fpu service.

The e$synch_channel_with_fpu service does the following:

1. From channel body, get pointer to FPU object body
2. Get sequence number from FPU object body
3. Update channel's sequence number the previous FPU sequence number.

The format of e$synch_channel_with_fpu is as follows:

```c
PROCEDURE e$synch_channel_with_fpu (  
    IN channel: POINTER e$channel_object_body;  
);

++
|
| Routine description:
| |
| This routine synchronizes the channel sequence number with that of
| the FPU.
| |
| Arguments:
| |
| channel    reference pointer to channel object.
| |
| Return value:
| |
| none
| |
|--
```
8.4.5 IRP Processing Executive Services

This section describes the services and procedures used to manipulate an I/O Request Packet (IRP). Listed below are the IRP services defined by the I/O architecture.

<table>
<thead>
<tr>
<th>Service</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>e$allocate_fp_parameter_record</td>
<td>Allocate a fixed amount of space in IRP for FP parameter record</td>
</tr>
<tr>
<td>e$deallocate_fp_param_record</td>
<td>Deallocate space in IRP for FP parameter record</td>
</tr>
<tr>
<td>e$initialize_irp</td>
<td>Initialize I/O request packet</td>
</tr>
<tr>
<td>e$extend_fp_parameter_record</td>
<td>Extends an FP parameter record</td>
</tr>
<tr>
<td>e$allocate_irp</td>
<td>Allocate an I/O request packet</td>
</tr>
<tr>
<td>e$deallocate_irp</td>
<td>Deallocate an I/O request packet</td>
</tr>
<tr>
<td>e$get_size_of_htl</td>
<td>Return size of HTL data structure</td>
</tr>
</tbody>
</table>

8.4.5.1 FP Parameter Record Allocation and Deallocation

Function processors are constantly passing an IRP from one function processor to another while processing an I/O request. In order for some lower level function processors to process a request, the upper level function processors create an FP parameter record and initialize it with the appropriate parameters. The I/O architecture provides executive service routines to manage FP parameter records within an IRP. A detailed description of these services is provided in the following sections.

8.4.5.1.1 e$allocate_fp_parameter_record

The e$allocate_fp_parameter_record service allows function processors to allocate an FP parameter record within the free area of an IRP. This service creates an FP parameter record, provided there is enough space, and initializes it. This service completes synchronously.

The format of the e$allocate_fp_parameter_record service is as follows:

```c
PROCEDURE e$allocate_fp_parameter_record ( 
    OUT fp_record: POINTER e$fp_parameter_record; 
    IN irp: POINTER e$io_request_packet; 
    IN record_size: e$unsigned; 
); 
EXTERNAL;
```

```c
+++ 
| | 
| Routine description: | 
| | This is an I/O Architecture executive service to allocate an fp parameter record in the irp. |
| | 
| Arguments: | 
| | fp_record Pointer to the newly allocate fp parameter record |
| | irp pointer to the irp |
| | record_size size of the fp parameter record |
| | 
| | Return value: |
| | none |
|---

8–88 I/O ARCHITECTURE
The $allocate_fp_parameter_record$ service does the following:

BEGIN $allocate_fp_parameter_record$
    Establish condition handler
    Get current size of IRP free area
    IF record_size > current size of IRP THEN
        Set fp_record to NIL
        raise condition - reason: Unable to allocate space in irp
    ELSE
        allocate space in IRP free area
        Store previous fp parameter record in header
        place address of allocated space in out fp_record
    ENDIF
END $allocate_fp_parameter_record$

8.4.5.1.2 $deallocate_fp_param_record$

The $deallocate_fp_param_record$ service is provided to allow the function processors and the I/O architecture to deallocate the most recently created FP parameter record. This service completes synchronously.

The format of $deallocate_fp_param_record$ is as follows:

PROCEDURE $deallocate_fp_param_record$
    (IN irp: POINTER $e$io_request_packet;
    )
    EXTERNAL;
++
|
Routine description:
|
| I/O architecture executive service to deallocate the current FP
| parameter record.
|
Arguments:
|
| irp IRP to used for deallocating fp parameter record
|
Return value:
|
| none
|
--

The $deallocate_fp_param_record$ service does the following:

BEGIN $deallocate_fp_param_record$
    Establish condition handler
    IF internal book keeping pointers are valid THEN
        Return allocated record to the IRP free space
    ELSE
        Raise condition - Invalid back pointers in parameter record
    ENDIF
END $deallocate_fp_param_record$
8.4.5.1.3 e$extend_fp_parameter_record

The **e$extend_fp_parameter_record** service allows a function processor to extend the current parameter record.

The format of **e$extend_fp_parameter_record** is as follows:

```c
PROCEDURE e$extend_fp_parameter_record (
   IN irp: e$sio_request_packet
   IN fp_record_size: e$unsigned;
);
EXTERNAL;
```

```
+++  
!  ! Routine description:
!  ! This routine allows the function processor to extend the size
!  ! of the current fp parameter record. The new parameter specifies the new
!  ! size of the fp parameter record.
!  ! Arguments:
!  ! irp       pointer to IRP
!  ! fp_record_size new total size of fp parameter record.
!  ! Return value:
!  ! none
!  ! -- 
```

8.4.5.1.4 e$allocate_irp

The **e$allocate_irp** service creates a I/O request packet (IRP). This routine will usually be called by the function processor's Initialize I/O Parameters procedure. As stated earlier, the function processor initially allocates an IRP via a call to this service. This procedure allows the function processor to dynamically allocate memory from non-paged pool for an IRP on a per I/O request basis. Any additional space needed by the function processor can be allocated as part of the IRP. This eliminates an extra call by the function processor to the memory pool allocator. See the Chapter 7, Memory Management for more details.

The format of the **e$allocate_irp** service is as follows:

```c
PROCEDURE e$allocate_irp ( 
   OUT irp: POINTER e$sio_request_packet;
   IN size_of_fp_parameter_area: e$unsigned; 
);
EXTERNAL;
```

```
+++  
!  ! Routine description:
!  !    I/O architecture executive service for allocating an IRP.
!  ! Arguments:
!  !    irp       pointer to the newly allocated IRP 
```
8.4.5.1.5 e$deallocate_irp

The e$deallocate_irp service deallocates an I/O request packet. This service provides the function processor that has just recently created an IRP with the capability to delete the IRP if an error condition occurs. This routine is also called by the e$complete_io service.

The format of e$deallocate_irp is as follows:

```
PROCEDURE e$deallocate_irp (  
   IN irp: POINTER e$io_request_packet;  
);  
EXTERNAL;

+++  
!
Routine description:
  
   This routine deallocates a previously allocated IRP.

Arguments:
  
   irp        pointer to irp to be deallocated

Return value:
   
   none

---
```

8.4.5.1.6 e$get_size_of_htl

The e$get_size_of_htl service returns the size needed for allocating an HTL.

The format of e$get_size_of_htl is as follows:

```
PROCEDURE e$get_size_of_htl (  
   IN buffer_starting_address: POINTER anytype;  
   IN buffer_length: e$unsigned;  
) RETURN htl_size;
```
8.4.6 Synchronizing I/O

This section describes the services used to synchronize various operations with an I/O request.

Table 8-12: Synchronizing I/O Services

<table>
<thead>
<tr>
<th>Service</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>exec$synchronize_io</td>
<td>Synchronize with completion of previously-issued I/O request</td>
</tr>
<tr>
<td>e$synchronize_io</td>
<td>Executive form of exec$synchronize_io</td>
</tr>
<tr>
<td>e$complete_io</td>
<td>Handles completion details of an I/O request</td>
</tr>
<tr>
<td>e$lock_io_buffer</td>
<td>Lock I/O buffer for direct I/O</td>
</tr>
<tr>
<td>e$unlock_io_buffers</td>
<td>Unlock I/O buffers</td>
</tr>
</tbody>
</table>

8.4.6.1 exec$synchronize_io

The exec$synchronize_io service can be used to synchronize with the completion of a previously issued I/O request. For example, if a thread has issued several asynchronous I/O requests and later wants to synchronize with one of them, it may do so by calling exec$synchronize_io, if an event object was specified on the request. The exec$synchronize_io service frees the user from having to write his own synchronization routines when performing asynchronous I/O. When control is returned from this service, the I/O is completed.

This service takes an event object ID and an IOSB as input. If the IOSB is already nonzero, the exec$synchronize_io completes immediately, without ever going to kernel mode. If the IOSB currently contains zero, then a wait occurs for the specified event object via a kernel wait call.

The format of the exec$synchronize_io service is as follows:

```{}
PROCEDURE exec$synchronize_io ( 
    IN user_event_id: exec$object_id; 
    IN user_iosb: POINTER exec$iosb; 
) RETURNS STATUS; 
EXTERNAL;
```
8.4.6.2  e$synchronize_io

The e$synchronize_io service was primarily written to be used internally by the exec$request_io service in order to allow a reliable implementation of synchronous I/O. The e$synchronize_io service should not be required for normal user software, unless, for example, an event object is shared between AST and non-AST processing, which is a bad idea.

The e$synchronize_io service is called by exec$synchronize_io. The e$synchronize_io service may also be called by kernel code to synchronize the completion of a previously issued I/O request when executive event objects are used. The synchronization is performed by waiting on the supplied event object to be signaled. The event parameter must be the same as that signaled by e$complete_io when the I/O request has completed. When the event has been signaled, the e$synchronize_io service determines if the I/O request has been completed by checking the condition field in the IOSB. If this field contains a value other than zero, the I/O request has been completed. The condition field in the IOSB gets changed by the e$complete_io service.

Below is the format of the e$synchronize_io service:

```c
PROCEDURE e$synchronize_io (  
    IN user_event: e$object_id;  
    IN user_iosb: POINTER e$iosb;  
)  
  RETURNS STATUS;  
  EXTERNAL;
```

```
+++  
! Routine description:  
! I/O architecture executive service interface for synchronizing with  
! a previously issued I/O request.  
! Arguments:  
! user_event object id of event object to signal when the  
! io completes  
! user_iosb I/O status block  
! Return value:  
! none  
!--
```
The `e$synchronize_io` service routine does the following:

```
BEGIN e$synchronize_io
  Probe parameters
  IF event object is not valid THEN
    set status - Invalid event object
  ELSE
    Get reference pointer of event object
    Get mode
    call e$synchronize_io_trusted(mode, event, iosb)
  ENDIF
END
```

8.4.6.2.1 `e$synchronize_io_trusted`

The `e$synchronize_io` service can also be called by kernel code to synchronize the completion of a previously issued I/O request when kernel events are used. This synchronization is performed by waiting on the supplied event object to be signaled. The event parameter must be the same as that signaled by `e$complete_io` when the I/O request completes. In other words, the event must be the same event that was used in the request I/O service.

The `e$synchronize_io_trusted` service is called by `e$request_io_trusted` to perform synchronous I/O. This service uses the IOSB to determine which I/O request to synchronize with. An optional event object is specified when the call originates from user mode via `exec$synchronize_io`. When the call originates from kernel mode, the I/O dispatcher's event object in the thread's TCB is used. The synchronization is performed by waiting on the event object to be signaled by the `e$complete_io` service.

The format of `e$synchronize_io_trusted` is as follows:

```
PROCEDURE e$synchronize_io_trusted (  
  IN mode: k$processor_mode;  
  IN iosb: POINTER e$iosb;  
  IN user_event: e$object_id = DEFAULT;  
  IN kernel_event: Pointer k$dispatcher_object(event) = DEFAULT;  
) RETURNS STATUS;  
EXTERNAL;
```

```++
! Routine description:
!
! This routine synchronizes an I/O request that has been initiated
! from either user or kernel mode.
!
! Arguments:
!
! mode caller's mode
! iosb Pointer to IOSB
! user_event user event object id.
! kernel_event reference pointer to kernel event object
!
! Return value:
!
! TBS
!
--
```

The `e$synchronize_io_trusted` service does the following:
BEGIN e$synchronize_io_trusted
WHILE I/O status block's condition field is NOT zero LOOP
  IF mode equal k$kernel AND kernel event object not present THEN
    call k$wait_single with I/O event object in thread's TCB
  ELSE IF mode is k$kernel THEN
    call k$wait_single with kernel event object.
  ELSE
    do an executive wait on user event object.
  ENDIF
  IF error occurred in wait THEN ! IOSB condition field contains a zero
    do exception processing ! The event object was signaled to
  ENDIF
END LOOP
set status ! notify the waiting thread
END e$synchronize_io_trusted

8.4.7 I/O Completion Functional Interface and Design

A successfully requested I/O can be completed by the initial procedure-based code in the function processor, or by a system thread initiated by the function processor. An I/O request cannot be completed by an interrupt procedure. An interrupt procedure must notify a thread to complete the request, normally by setting a kernel event object.

An I/O request is completed by calling e$complete_io with the IRP and completion status as input.

8.4.7.1 e$complete_io

The primary task of the e$complete_io service is to store the completion status in the IRP and queue a special kernel-mode AST to the issuing thread. The bulk of I/O completion is done in the issuer’s context. Unlike other ASTs, special kernel-mode ASTs actually run at the priority of AST delivery level (IPL 1) to block all subsequent ASTs for their duration.

The format for e$complete_io is as follows:

PROCEDURE e$scomplete_io (  
  IN irp: POINTER e$io_request_packet;  
  IN completion_status: longword;  
  IN byte_count: longword = "0"x;  
  IN fp_data: quadword;  
  IN ast_output_parameter: e$ast_parameter;  
);  
EXTERNAL;

++
! Routine description:
! This is an I/O architecture executive service provided to complete io.
! Arguments:
! irp pointer to irp
! completion_status completion status
! byte_count data transfer count
! fp_info filled in by fp, then placed in iosb
! ast_output_parameter AST output parameter
!
Return value:
none
--
Since page faults are also allowed at IPL 1, special handling is required for the completion of in-page I/O requests in order to avoid deadlock (see Section 8.4.7.3 for more details). In-page I/Os cannot complete like normal I/Os, since the thread is already running at IPL 1 and all ASTs are blocked. This deadlock is avoided by always completing page fault I/Os directly in the $complete_io$, instead of queuing the special kernel-mode AST. For a complete description of how paging I/O is handled, see Section 8.4.7.3.

The execution of $complete_io$ is summarized as follows:

1. Write status, byte_count and FP_dependent input parameters to current_status in the IRP.
2. If direct I/O and request is successful, then unlock direct I/O buffers. (See Section 8.4.7.2.)
3. If in-page I/O (in_page field of IRP is TRUE), then do paging I/O completion. (See Section 8.4.7.3.)
4. If this was not an in-page I/O, then queue special mode kernel ast to finish I/O completion.

8.4.7.1.1 Special Kernel-Mode AST Processing

The special kernel-mode AST for I/O completion has primarily two responsibilities: First, it calls the function processor completion procedures recorded in the parameter records of the IRP in LIFO order. Secondly, it handles the final I/O completion of the user request, as dictated by the user-specified IOSB, event object, and AST procedure.

The first loop of the special kernel-mode AST is as follows:

```plaintext
PROCEDURE io$cast_complete_io
  IN irp: POINTER e$request_io_packet;
);
BEGIN io$cast_complete_io
  Clear channel pointer field in IRP
  WHILE current_parameters <> NIL LOOP
    IF current_parameters^completion_procedure <> NIL AND
      (NOT current_parameters^call_on_error_only OR even(status))
      THEN
        Call completion procedure with IRP and current_parameters
        current_parameters = current_parameters^previous;
      ENDIF;
    END LOOP;
call io$complete_io_cleanup(irp)
END io$cast_complete_ast
```

At the end of each loop iteration, a function processor parameter record is popped from the stack and returned to the free area of the IRP. The function processor's completion procedure must either return or raise the exec$sio_continued condition to indicate that it is continuing to process the I/O request. This essentially aborts I/O completion and dismisses the special kernel-mode AST to allow continued processing on the request. In this case, the issuer never sees a thing, since user-I/O completion has not been done.

If the WHILE loop completes, the user I/O request is completed as follows:

```plaintext
PROCEDURE io$complete_io_cleanup (
  IN irp: POINTER e$request_io_packet;
);
BEGIN io$complete_io_cleanup
```
IF IOSB_address in irp <> NIL THEN
  IOSB_address = current status; ! Ignore access violations
ENDIF;
IF event_pointer in irp <> NIL THEN
  k$signal_event(event_pointer, priority_boost);
  edecrement_pointer_count(event_pointer);
ELSE IF kernel_event_pointer in irp <> NIL THEN
  e$set_event_trusted(user event pointer)
ENDIF;
IF irp release_channel variable is TRUE THEN
  dereference channel
ENDIF;
IF AST_procedure in irp <> NIL THEN
  k$insert_ast_queue(AST_procedure, AST_parameter, io$kast_complete_io,
   irp, thread object_pointer);
  ! Piggy-back AST used to deallocate_IRP
ELSE
  call remove_irp(irp) procedure;
ENDIF;
END io$kcomplete_io_cleanup

The Remove_irp procedure is as follows:

PROCEDURE io$kremove_irp (
  IN irp: POINTER e$io_request_packet;
);
BEGIN io$kremove_irp
Remove IRP from thread object queue;
IF TCB io cancel flag is set THEN    ! Synchronizes I/O cancellation with thread
  IF TCB irp list is empty THEN    ! rundown.
  set TCB I/O cancel complete event
ENDIF
Acquire channel object mutex;
Remove IRP from channel queue;
Release channel object mutex;
IF the channel's outstanding IRP list is empty THEN
  get IRP pointer to deaccess packet and function code in channel object;
  call e$execute_io(channel, function, irp, access mode)
ENDIF
END io$kremove_irp

8.4.7.1.1 Function Processor's Complete I/O Procedure

As described earlier, a call to e$complete_io ultimately results in the execution of a special kernel-
mode AST in the context of the thread which issued an I/O request. This AST procedure traverses
the stack of parameter records in the IRP in LIFO order, popping parameter records as it goes. For
each parameter record, if the function processor has loaded the address of a completion procedure,
either an error status is being returned or the function processor wishes to be called on all
completions (call_on_error_only is FALSE), then this completion procedure is called.

The executive type definition for the Completion procedure is as follows:

TYPE
  e$completion_procedure :
  PROCEDURE (
    IN fp_parameter_record: e$fp_parameter_record;
    IN irp: POINTER e$io_request_packet;
  );

The function processor defines its Completion procedure as follows:
PROCEDURE completion_procedure (  
fp_parameter_record,  
irp  
) OF TYPE e$complete_io_procedure;

++
!
! Routine description:
!
! This is the interface definition for the function processor completion  
procedure.
!
! Arguments:
!
! fp_parameter_record     pointer to function processor parameter in irp  
irp     pointer to IRP
!
! Return value:
!
! none
!
--

Essentially the function processor completion procedure must do one of two things:

1. It can retry or continue processing on the I/O request by processing the IRP itself, or by passing it  
to another function processor via a call to e$execute_io. An example of this would be the shadow  
function processor retrying a failed read by passing the IRP via e$execute_io to another member  
of the shadow set. If the completion procedure decides to retry or continue the request, then  
instead of returning, it must raise the exec$io_continued condition.

2. Alternately, the completion procedure may simply do some cleanup and return. Some of this  
cleanup may only be relevant if the request was successful. Other cleanup may be necessary,  
regardless of the completion status of the request. Naturally the cleanup operations performed by  
the completion procedure must be totally driven off of fields in the function processor's parameter  
record.

Following are examples of the most common cleanup activities performed in the completion procedure:

- Dereferencing pointers. Pointers that were acquired when the request was issued must be re-  
released.

- Buffered input copy. For a buffered input, the final copy to the user's I/O buffer may take place.  
Page faults during this process are allowed. The buffered input is normally present in a buffer  
pool managed by the function processor. After the copy, this buffer must be freed to the function  
processor pool. (For a discussion of buffered output, see Section 8.4.1.1.2).

- Modifying error status. If an error is being returned from a lower-level function processor, it is  
possible that the error code being returned is not valid for the current function processor. In  
this case, the error status may be modified accordingly. For example, a virtual disk function  
processor may ultimately make a request to the Network Interconnect (NI) function processor,  
yet most errors defined for NI are not valid when returned from a disk function processor.

- Other. A function processor may have other specific cleanup operations to perform. An example  
might be the volume transaction count for the Files–11 VFP.
8.4.7.2 Direct I/O Buffers

Direct I/O buffers are buffers into which a direct-connected device transfers data directly by physical address. Since direct-connected devices use physical addresses, and since they are totally unsynchronized with the Mica memory management support, it is necessary to lock in memory any buffers required by these devices for the duration of the request.

When the I/O request completes, it is desirable to unlock the buffers as soon as possible, and thus return them to the pool of pages where memory management can do whatever it wishes with them. As described, the normal I/O completion mechanism operates via a special kernel mode AST in the issuing thread. However, it is now possible that something has happened to the issuing thread so that it may not be eligible for execution for an indefinite period of time. Therefore, it is very desirable to be able to unlock the direct I/O buffers prior to queuing the I/O completion AST for the thread. For example, in the worst case, the balance set scheduler may have removed the thread's process from the balance set in an attempt to release all of the process memory. We don't want to wait for a special kernel-mode AST to execute in the issuing thread before we can free its I/O buffers!

To allow the system to free direct I/O buffers quickly outside of the normal I/O completion processing in the issuing thread, a standard service called \texttt{e$unlock\_buffer} is provided by memory management. For pre-processing direct I/O buffers during I/O parameter processing, memory management has provided a standard service called \texttt{e$probe\_and\_lock\_buffer}. This service initializes a data structure called the host transfer list (HTL) and locks the buffer down in memory.

When a request completes with success, \texttt{e$complete\_io} will unlock all direct I/O buffers in the HTL and clear the HTL listhead in the IRP. On error, the HTL(s) will remain locked so that a completion procedure can retry the request, if desired. If \texttt{e$request\_io} cannot acquire the memory management database mutex, it will pass the IRP to a special system thread, which will do the unlocking. If the HTL listhead is nonzero when all completion procedures have been called in the special kernel mode AST for I/O completion, then the buffers will be unlocked at that time. This can only occur when a request gets an error and no completion routines attempt to retry the I/O. Because of the way direct I/O buffers are unlocked, a completion procedure can only reissue a direct I/O when an error has occurred.

8.4.7.2.1 \texttt{e$lock\_io\_buffer}

The \texttt{e$lock\_io\_buffer} service initializes the HTL, links the HTL into the IRP, and calls a memory management routine (\texttt{e$probe\_and\_lock\_io\_buffer}) to lock buffers as described by the HTL in memory. Since multiple HTLs may be linked together, the \texttt{e$lock\_io\_buffer} service only locks down one buffer at a time.

The \texttt{e$lock\_io\_buffer} routine takes an input parameter, a pointer to an IRP, and a pointer to an HTL. If multiple HTLs are linked together before calling this service, then the first HTL in the linked list must be supplied as input to this service. The format of the \texttt{e$lock\_io\_buffer} service is as follows:

\begin{verbatim}
PROCEDURE e$lock_io_buffer(
    IN  irp: POINTER e$io_request_packet;
    IN  htl: POINTER e$host_transfer_list;
    IN  buffer_starting_address: POINTER anytype;
    IN  buffer_length: e$unsigned;
);
\end{verbatim}

8.4.7.2.2 \texttt{e$unlock\_io\_buffers}

The \texttt{e$unlock\_io\_buffers} service is used by a function processor to unlock previously locked I/O buffers. The format of the \texttt{e$unlock\_io\_buffers} service is as follows:

\begin{verbatim}
PROCEDURE e$unlock_io_buffers(
    IN  irp: POINTER e$io_request_packet;
);
\end{verbatim}
8.4.7.3 Paging I/O

Special treatment is provided for in-page I/O to allow page faults to occur at IPL 1. Without this special treatment, it would be impossible for code running at IPL 1 to get a page fault.

The I/O architecture has reserved two I/O function codes for use by memory management: io$c_page_read and io$c_page_write. These function codes are used to facilitate paging I/O. When the file system function processor receives either of these function codes, it must change them to a normal read or write function code. At this point, the HTL would have already been initialized by memory management. Memory management passes a pointer to the HTL as a part of the request I/O parameters. This alleviates the need for the file system function processor to build the HTL and call the e$probe_and_lock_buffer service. However, the file system function processor must link the HTL into the IRP. Since memory management allocated storage for the HTL, it is responsible for deallocating the HTL after I/O completion.

A file system supporting mapped files must also support the io$c_page_read and io$c_page_write function codes from kernel mode. These function codes require that the in_page boolean in the IRP be set to TRUE, and the function code be changed to that of a normal read request. One exception is that the input buffer does not need to be locked down.

On completion of a successful in-page operation, the IRP parameter stack is processed and all of the completion procedures are called directly by e$complete.io. On error, the IRP is passed to a special system thread that processes the parameter stack.

The completion procedures called for page operations cannot get page faults and cannot count on running in the issuing thread's context. Note that this does not include any page faults initiated by the file system.

The I/O architecture has provided an internal procedure called io$complete_paging_io to facilitate the completion of paging.io. This procedure is called by e$complete.io if the request was an in-page I/O. The io$complete_paging_io procedure is executed, instead of queuing a special kernel mode AST. This procedure then completes the I/O request in the context of the calling thread.

The format of the io$complete_paging_io function code is as follows:

```c
PROCEDURE io$complete_paging_io(
    IN irp: POINTER e$request_io_packet;
);
```

The io$complete_paging_io procedure does the following:

```c
BEGIN io$complete_paging_io
    call io$ast_complete_io(irp) procedure
    call io$complete_io_cleanup(irp) procedure
END io$complete_paging_io
```

8.4.8 Canceling I/O Requests

This section describes the services used to cancel an I/O operation.

<table>
<thead>
<tr>
<th>Table 8-13: Cancel I/O Request Services</th>
</tr>
</thead>
<tbody>
<tr>
<td>Service</td>
</tr>
<tr>
<td>exec$cancel_io</td>
</tr>
<tr>
<td>e$cancel_io</td>
</tr>
<tr>
<td>e$cancel_io_by_thread</td>
</tr>
<tr>
<td>e$cancel_io_by_channel</td>
</tr>
<tr>
<td>e$insert_cancel_procedure</td>
</tr>
<tr>
<td>e$remove_cancel_procedure</td>
</tr>
</tbody>
</table>

8-100  I/O ARCHITECTURE
Outstanding I/O requests can be canceled through the channel object to which they are queued either by an explicit user call, or by object rundown on a channel being deleted. I/O requests can also be canceled through the thread object by the system during thread rundown. There is no mechanism to cancel individual I/O requests from user mode.

Internally, requests are canceled individually. Requests are canceled by calling a cancellation procedure whose address has been stored in the `cancel_procedure` field in the IRP. This field is filled in by a function processor that owns the IRP and wishes to support cancellation over a given period of time.

The Cancel request is always completed immediately. However, since request cancellation completes asynchronously, it is still necessary for user and system software to synchronize with the original requests in order to know when they are done.

There is no provision for canceling an I/O request in a function processor unit request queue. Therefore, it is the responsibility of the function processor to make sure that the unit never hangs in such a manner to cause an IRP to remain in the queue for an indefinite amount of time. Such a condition makes it impossible to cancel an I/O request. For example, this condition may potentially cause a thread to wait forever if it is waiting for an I/O request to be canceled. See Section 8.4.8.1.1 for more information on cancellation.

### 8.4.8.1 exec$cancel_io

For each outstanding request on a channel object, an attempt is made to cancel the request.

The format of the `exec$cancel` call is as follows:

```java
PROCEDURE exec$cancel_io (    
    IN channel_id: exec$object_id;  
    ) RETURNS STATUS;  
EXTERNAL;
```

```
!++
! Routine description:
! I/O architecture system service to cancel all outstanding I/O on a channel.
! Arguments:
! channel_id channel object id, for the cancel operation
! Return value:
! TBD
!--
```

The `e$cancel_io` service routine does the following:

```java
BEGIN e$cancel_io
    Probe parameters
    Get pointer to channel object body
    call e$cancel_io_by_channel
    set status
END e$cancel_io
```

I/O ARCHITECTURE 8-101
8.4.8.1.1 Function Processor’s Cancel I/O Request Procedure

The Cancel I/O procedure cancels an I/O request. Although the internal mechanism described here is for canceling a single request, there is no interface available to users to cancel individual I/O requests. This is because I/O requests are not objects. Therefore, the caller is not given any “handle” to identify individual I/O requests. Cancel is meant to be a cleanup operation for exceptional conditions, and not to be used as part of normal programming.

The executive type definition for the Cancel I/O Request procedure is as follows:

```c
TYPE
  e$cancel_io_procedure:
    PROCEDURE (
      IN irp: POINTER e$io_request_packet;
    );
```

The function processor defines its Cancel I/O Request procedure as follows:

```c
PROCEDURE cancel_io (
  irp
) OF TYPE e$cancel_io_procedure;
```

```cpp
++
|
| Routine description:
|
| This is the function processor procedure that is used to cancel
| an I/O request.
|
| Arguments:
|  | irp               pointer to irp
|
| Return value:
|  | none
|--
```

Some function processors may choose not to support Cancel. The types of requests that are most important to be able to cancel are any operations that can take an indefinite amount of time to complete. Such operations include, DECnet receives, input requests or blocked output requests on pipes, and so on.

Cancel calls are dispatched through a cancel flag and a procedure pointer in the IRP. By default, the cancel flag is initialized to FALSE, and the pointer is initialized to no-op the Cancel call; that is, the request cannot be canceled. Whenever a function processor places a request in a state where it may take an indefinite amount of time to complete, such as in those cases previously mentioned, then it should call e$insert_cancel_procedure to fill in the Cancel procedure pointer. Attempts to cancel a request set the cancel flag to TRUE and call the Cancel procedure.

Calls to the Cancel procedure, and modifications to the cancel flag and Cancel procedure pointer, are synchronized by a mutex contained in the channel object. Whenever a new Cancel procedure address is written into the IRP, the flag is tested. If the cancel flag is TRUE, the Cancel procedure is called immediately. Function processors establishing Cancel procedures must be prepared to immediately handle a call to the procedure from the point at which it is established to the point at which the Cancel procedure is cleared.

The Cancel procedure is automatically cleared by e$complete_io when the I/O is completed, as well as by e$execute_io when an IRP is passed on to another function processor.

The following discussion centers around what rules a function processor should follow in supporting calls to the Cancel procedure.

---

8–102 I/O ARCHITECTURE
If the IRP is queued within the function processor and it is not currently being serviced by a system thread, then it is acceptable to complete the IRP directly with an abort status. This operation must be synchronized by the channel mutex and the function processor's request queue mutex in order to serialize the Cancel with potential completers of the request.

8.4.8.1.1 Manipulating an IRP on the Request Queue

Whenever an IRP is inserted on or removed from the function processor's request queue, the address of the function processor's Cancel procedure is respectively written into or cleared from the IRP.

When an IRP is added to a function processor's request queue, a Cancel procedure should be inserted in the IRP. This allows an I/O request to be canceled when the IRP is currently in a function processor's request queue.

Below is a basic description for inserting an IRP into a function processor's request queue and inserting the cancel procedure into the IRP.

```
BEGIN fp$insert_irp_in_request_queue
  Get channel pointer from IRP
  Acquire channel mutex
    Acquire FP request queue mutex
    Insert IRP into request queue
    Insert cancel procedure into IRP (via )
    Release FP request queue mutex
  Release channel mutex.
END fp$insert_irp_in_request_queue
```

When removing an IRP from the function processor's request queue, the cancel procedure should be cleared. Synchronization to the function processor's request queue and the IRP is provided when the mutex for the request queue and the channel has been acquired.

Below is a description of how an IRP is removed from a function processor's request queue and of how the cancel procedure is cleared in the IRP.

```
BEGIN fp$remove_irp_in_request_queue
  Acquire FP's request queue mutex
    Remove IRP from request queue
  Release request queue mutex
  Remove cancel procedure from IRP. This is done
    by calling (irp)
END fp$remove_irp_in_request_queue
```

When a function processor's Cancel procedure is called, it must determine if the IRP is in a request queue. If so, the IRP must be removed from the queue, the cancel procedure must be cleared, and the request must be completed. The request may be completed by calling `e$complete_io` with a pointer to the IRP as an argument, or it may be canceled. However, if the IRP is not in a request queue, then the request queue mutex is released and the Cancel procedure exits.

Below is description of what the cancel procedure must do when a request is canceled.

```
BEGIN fp$cancel Procedure
  (Note: The channel mutex is currently being held prior
  to the call of the FP's cancel procedure.)
  Acquire FP's request queue mutex
  IF IRP is still in request queue THEN
    remove IRP from request queue
  Release FP's request queue mutex
  Perform any FP specific cancel procedures
    complete request. (via )
  ELSE
    Release FP's request queue mutex
  ENDIF
```

I/O ARCHITECTURE 8-103
END fp$cancel_procedure

Also note that the channel mutex is at a higher level than the request queue mutex. This means that, if an attempt is made to acquire both the channel and the request queue mutex simultaneously, then the channel mutex must be acquired first.

If the IRP is currently being processed by a system thread, then the Cancel procedure is more complicated. For reasons of synchronization and of getting the system thread back to the right state, it is generally a bad idea to try to cancel the IRP directly. The recommended procedure for canceling an IRP being processed by a system thread is to force an event that causes the system thread to cancel the request itself. Two examples of how this might be done are as follows:

- If a function processor system thread servicing the IRP is currently waiting on an I/O request of its own that may not complete soon, the I/O request can be canceled to cause the thread to unwait and notice that its packet is to be canceled.
- If the thread servicing the IRP is currently waiting for an interrupt that may not complete soon, the event associated with the interrupt can be signaled to cause the thread to continue and detect that the IRP is to be canceled.

The above are only examples of possible strategies; they are meant to indicate the type of solution that can be applied. The actual solution for any given function processor must be carefully designed.

The Cancel request itself is always completed synchronously, but it is still necessary for user and system software to synchronize with the original requests to know when they are done, since their cancellation will complete asynchronously.

8.4.8.2 e$cancel_io_by_thread

The e$cancel_io_by_thread service must be called in the context of the thread belonging to the I/O request to be canceled. This service completes synchronously.

The format of the e$cancel_io_by_thread service is as follows:

```
PROCEDURE e$cancel_io_by_thread (  
);
EXTERNAL;
```

```
!++
! Routine description:
! This is the I/O architecture executive service to cancel all outstanding I/O for this thread.
! Arguments:
! thread     thread object id
!
! Return value:
! none
!--
```

The e$cancel_io_by_thread service routine does the following:
BEGIN e$cancel_io_by_thread

Get reference pointer to current thread
Set TCB cancel I/O flag
Clear TCB complete I/O event
Disable kernel mode ASTs
While the thread’s I/O request queues is not empty
LOOP
  IF there is a valid channel pointer in IRP THEN
    acquire the channel mutex
    IF channel pointer in IRP is still valid and I/O has not been canceled THEN
      IF there is a cancel procedure in IRP THEN
        call cancel procedure
        remove cancel procedure from IRP
        (via e$remove_cancel_procedure)
      ENDIF
    ENDIF
    Release Channel Mutex
  ENDIF
END LOOP
Enable Kernel-mode ASTs
Wait on TCB cancel io complete event

END e$cancel_io_by_thread

8.4.8.3 e$cancel_io_by_channel

The cancel I/O by channel (e$cancel_io_by_channel) service cancels I/O by channel. This service completes asynchronously.

The format of e$cancel_io_by_channel is as follows:

PROCEDURE e$cancel_io_by_channel ( IN channel: POINTER e$channel_object_body;
  );
EXTERNAL;
+++
| Routine description:
| This is an I/O architecture executive service routine to cancel outstanding I/O requests on a channel.
| Arguments:
| channel reference pointer to channel object
| Return value:
| none
|--

The e$cancel_io_by_channel service routine does the following:

BEGIN e$cancel_io_by_channel
IF the channel I/O request queue is not empty THEN
   Acquire channel mutex
   Get Current thread pointer
   WHILE the channel I/O request queues is not empty
      LOOP
         IF the thread pointer in IRP is for current thread and I/O
            has not been canceled THEN
            IF there is a cancel procedure in IRP THEN
               call cancel procedure
               remove cancel procedure from IRP
               (via e$remove_cancel_procedure)
            ENDIF
         ENDIF
      ENDLOOP
   Release Channel Mutex.
ENDIF
END e$cancel_io_by_channel

8.4.8.4 e$insert_cancel_procedure

The insert cancel procedure (e$insert_cancel_procedure) service copies the function processor cancel
procedure into the IRP. A check is also made to determine if this I/O request was marked cancel. If
the I/O request is marked cancel, then the function processor cancel procedure is called via the cancel
procedure address stored in the IRP. This service completes synchronously.

The format of the e$insert_cancel_procedure service is as follows:

PROCEDURE e$insert_cancel_procedure (
   IN irp: POINTER e$io_request_packet;
   IN cancel_io_procedure: e$cancel_io_procedure;
);
EXTERNAL;

++
|!
| Routine description:
|!
| This is the I/O executive service routine to add the function processor
| cancel procedure to the IRP.
|!
| Arguments:
|!
| irp
| new_cancel_procedure
|!
| Return value:
|!
| none
|!
--

The e$insert_cancel_procedure service routine does the following:

BEGIN e$insert_cancel_procedure
   Acquire channel mutex    ! Pointer to channel is found in IRP
   Write new cancel procedure in the irp
   IF the cancel issued flag in the IRP is TRUE THEN
      call cancel procedure stored in IRP
      remove cancel procedure from IRP
   ENDIF
   Release channel mutex
END e$insert_cancel_procedure

8–106  I/O ARCHITECTURE
8.4.8.5  e$remove_cancel_procedure

The remove cancel procedure (e$remove_cancel_procedure) service clears the cancel_procedure field in the IRP. This service completes synchronously.

The format of the e$remove_cancel_procedure service is as follows:

PROCEDURE e$remove_cancel_procedure (  
  IN irp: POINTER e$io_request_packet;  
);  
EXTERNAL;

++
!
!  Routine description:
!
!  This is the I/O executive service routine to clear the IRP cancel_procedure
!  field.
!
!  Arguments:
!
!    irp            pointer to the IRP
!
!  Return value:
!
!    none
!
!--

The e$remove_cancel_procedure service routine does the following:

BEGIN e$remove_cancel_procedure

  Acquire mutex in channel  ! Pointer to channel is found in IRP
  Clear cancel procedure in IRP
  Release channel mutex

END e$remove_cancel_procedure

The e$insert_cancel_procedure and the e$remove_cancel_procedure can only be called by threads which currently own or are responsible for the IRP.

8.4.9  Startup and Initialization of the I/O Architecture

The I/O system is initialized by the io$initialize_io_arch I/O architecture routine. Initialization of the I/O system consists of the following steps:

- Allocate I/O static data block
- Create an FPD OTD object
- Create an FPU OTD object
- Create an CHANNEL OTD object
- Stored the OTD object ids in the I/O static data block

As listed above, the I/O architecture creates three object type descriptors (OTDs): the FPD OTD object, the FPU OTD object, and the channel OTD object. These OTD objects are placed in a pre-defined container during the creation process by the e$create_ots service provided by the object architecture. After creating these objects, the object architecture returns an object ID for each object. The OTD objects IDs are stored in a pre-allocated static storage block for use later by the I/O architecture when creating a channel object, FPU object, or an FPD object. The FPU OTD is created so that FPU objects can be waited on. For more information about OTD objects see Chapter 5, Object Architecture.
The format of the io$initialize_io_arch interface is listed below:

PROCEDURE io$initialize_io_arch(
);

8.4.10 Function Processor Startup and Initialization Procedure

Function processors are built as standard images. Function processors not required during system bootstrap and initialization are "loaded" into the system address space by means of a special loader. This loader then calls the image initialization procedure, which performs the required function processor initialization and returns.

Function processors required during system bootstrap and initialization are linked with the system. During system initialization, the function processor is called at its normal transfer address. The function processor must perform its required initialization and return to the caller when done.

Note that the function processor does not have to distinguish between the two cases of being linked with the system or being activated later; when called at its transfer address, it simply initializes and returns.

The function processor must do the following during its initialization:

- If the function processor supports any function-processor-specific objects, it must declare its object service routines to the system.
- If the function processor requires any pages to be locked in memory, it must perform the necessary system calls to do so.
- All function processors must declare themselves to the system by creating an FPD object. Among other things, the FPD contains a vector of addresses of standard procedures for the function processor. The FPD is named after the function processor. This name must normally be fixed, since it is specified by other software when a function processor unit is created.

If the function processor uses system threads, it normally does not create them as part of startup and initialization, but rather later as part of FPU creation.

The function processor may or may not create any FPUs during its initialization. Decisions to create FPUs may be external to the function processor, and result in calls to the function processor's Create FPU procedure.

8.4.10.1 FP & FPU Initialization and Creation Overview

The following sequence gives a quick summary of how function processors become known to the system, how FPUs are created, how subsequent channels to these units are created, and how I/O is performed:

1. Function processors are built as standard images.

2. The function processor is "loaded" into the system address space and called at its standard transfer address. Some function processors must be linked directly into the system boot image and be activated during system initialization.

3. Once called, the function processor initializes itself. As part of this initialization, it creates its FPD object. The FPD contains pointers to the standard function processor procedures which can be entered by the executive services and I/O architecture procedures described throughout this chapter. The FPD also contains the size of the FPU objects to be allocated for units supported by this function processor. The function processor may also lock some of its image pages into physical memory.

4. Next, a user program (such as MOUNT), or the system or function processor itself, issues an exec$create_fpu service, specifying that an FPU is to be created for the function processor.
5. The `exec$create_fpu` service creates the FPU object (according to the size in the FPD) and calls the function processor at its Create FPU procedure. The function processor initializes its specific data structure within the FPU for its type of unit. The function processor is now ready to accept channels for I/O requests on this unit.

6. The Create channel service is called, specifying the newly created FPU. A channel object is created and initialized, but the function processor is not called at this time.

7. At this point, a function-processor-specific Access function may be required to allocate and initialize an additional data structure required by the function processor to maintain additional context for the channel. If so, an Access request is issued on the newly created channel.

8. Now I/O requests are issued on this channel via Request I/O calls to the channel object. These calls are dispatched to the function processor via its FPD. The function processor receives all calls initially as procedure calls. It may propagate further execution as a system thread by queuing the request to a queue serviced by one or more threads.

8.5 Outstanding Issues

- I/O Performance—How will I/O performance be incorporated into the I/O architecture?
- Protecting `es$configure_fp` functions from user mode – How is this done?
- How function processors are loaded and initialized—How are function processors loaded into the system? How is the function processor initialize procedure called?