

Microkernel White Paper

Preliminary and Confidential

Macintosh System Software
Apple Computer, Inc.

Copyright (©) 1992-1995 Apple Computer, Inc.

ABOUT THE MICROKERNEL	8
KERNEL OBJECTS AND IDS.....	9
NAMING	11
EXECUTION - TASKING AND INTERRUPTS.....	12
ABOUT EXECUTION	12
ABOUT TASKS	13
ABOUT TASK SCHEDULING	14
ABOUT SOFTWARE INTERRUPTS	15
ABOUT PRIVILEGED EXECUTION.....	16
ABOUT SYNCHRONIZATION.....	16
ABOUT INTERRUPTS	17
KERNEL PROCESSES	19
ADDRESS SPACE MANAGEMENT	20
ABOUT ADDRESSING	20
THE SYSTEM 7 ADDRESSING MODEL	21
THE MICROKERNEL ADDRESSING MODEL.....	22
<i>Multiple Address Spaces</i>	22
<i>Areas</i>	22
<i>Paging</i>	23
<i>Global Areas</i>	23
<i>I/O Coordination</i>	24
<i>Addressing And Execution</i>	24
<i>Inter-Address Space Access</i>	24
<i>Code Fragment Manager Contexts</i>	25
SYNCHRONIZATION MECHANISMS	26
MESSAGING.....	28
MESSAGES	28
CLIENT-SERVER	28
TRANSACTIONS	28
MOVING DATA	29
PORTS AND OBJECTS	30
SENDING MESSAGES	31
RECEIVING MESSAGES	31
REPLYING TO MESSAGES	32
MESSAGE TYPES	32
CANCELING ASYNCHRONOUS MESSAGE OPERATIONS	33
LOCKING MESSAGE OBJECTS	34
TIMING & TIMERS	36

MEASURING ELAPSED TIME	36
SUSPENDING EXECUTION	36
ASYNCHRONOUS TIMERS	36
SUMMARY OF BENEFITS	38
THE MICROKERNEL AND COPLAND	38
USING THE MICROKERNEL API	40
CALLING CONVENTIONS	40
STACK SPACE	40
ADDRESSING	41
SOME BASIC TYPES	42
MISCELLANEOUS TYPES	42
PARAMETER BLOCK VERSIONS	42
DURATION	42
TIME	43
IDS	43
ITERATORS	44
ERRORS	46
GENERIC ERRORS	46
KERNEL PROCESS MANAGEMENT.....	47
CREATING KERNEL PROCESSES	47
DELETING KERNEL PROCESSES	48
OBTAINING THE CURRENT KERNELPROCESSID.....	48
OBTAINING INFORMATION ABOUT A KERNEL PROCESS	48
TASK MANAGEMENT	50
ABOUT TASK HIERARCHY	50
ABOUT TASK SCHEDULING	50
ABOUT TASK PARAMETERS AND RESULTS	51
ABOUT TASK TERMINATION	51
THE TASKING SERVICES	53
CREATING TASKS	53
TERMINATING A SPECIFIC TASK	56
OBTAINING THE ID OF THE CURRENT TASK	57
DETERMINING THE AMOUNT OF AVAILABLE STACK SPACE.....	57
OBTAINING INFORMATION ABOUT A TASK	57
SETTING A TASK'S EXECUTION PRIORITY	59
ITERATING OVER TASK IDS	61
EXCEPTIONS	62
ABOUT EXCEPTION HANDLERS	62
<i>Implementation Note:</i>	63

EXCEPTIONS WITHIN EXCEPTION HANDLERS	63
EXCEPTION HANDLER DECLARATIONS	63
INSTALLING EXCEPTION HANDLERS	64
SOFTWARE INTERRUPTS	65
CONTROLLING SOFTWARE INTERRUPTS	65
QUERYING THE LEVEL OF EXECUTION	65
SPECIFYING SOFTWARE INTERRUPTS	66
SENDING SOFTWARE INTERRUPTS	66
DELETING A SOFTWARE INTERRUPT	67
HARDWARE INTERRUPTS	68
ABOUT INTERRUPT HANDLERS	68
DESIGNATING INTERRUPT SOURCES	68
EXCEPTIONS CAUSED BY INTERRUPT HANDLERS	68
EXECUTION CONTEXT OF INTERRUPT HANDLERS	69
ARBITRATING FOR INTERRUPTS	69
PARAMETERS TO INTERRUPT HANDLERS	69
SECONDARY INTERRUPT HANDLERS	70
ABOUT SECONDARY INTERRUPT HANDLERS	70
EXCEPTIONS IN SECONDARY INTERRUPT HANDLERS	71
QUEUING SECONDARY INTERRUPT HANDLERS	71
CALLING SECONDARY INTERRUPT HANDLERS	72
EVENT FLAGS	73
CREATING EVENT FLAG GROUPS	73
DELETING EVENT FLAG GROUPS	73
SETTING EVENT FLAGS	73
CLEARING EVENT FLAGS	74
EXAMINING THE VALUE OF EVENT FLAGS	74
WAITING FOR EVENT FLAGS TO BECOME SET	74
THE PROCESSING OF SETEVENTS	75
KERNEL QUEUES.....	76
CREATING KERNEL QUEUES	76
DELETING KERNEL QUEUES	76
NOTIFYING A KERNEL QUEUE	77
NOTIFYING A KERNEL QUEUE FROM SECONDARY INTERRUPT LEVEL	77
WAITING ON A KERNEL QUEUE	78
KERNEL NOTIFICATION	80
KERNEL NOTIFICATION	80
KERNEL NOTIFICATION QUEUE	81
TIMING SERVICES.....	82
TIMER ACCURACY	82

ABOUT THE TIME BASE	82
SETTING THE TIME BASE	83
TIMING LATENCY	83
TIMER OVERHEAD	84
OBTAINING THE TIME	84
SETTING TIMERS TO EXPIRE IN THE PAST	84
SYNCHRONOUS TIMERS	84
<i>Synchronous Timers With Absolute Times</i>	84
<i>Synchronous Timers With Relative Times</i>	85
ASYNCHRONOUS TIMERS	85
RESETTING ASYNCHRONOUS TIMERS	86
INTERRUPT TIMERS	87
CANCELING ASYNCHRONOUS AND INTERRUPT TIMERS	88
ADDRESS SPACE MANAGEMENT	90
BASIC TYPES	90
<i>Static Logical Addresses</i>	91
ADDRESS SPACE CONTROL	91
<i>Creating Address Spaces</i>	92
<i>Deleting Address Spaces</i>	92
<i>Obtaining Information About an Address Space</i>	92
<i>Iterating Over All Address Spaces</i>	93
AREA CONTROL	94
<i>Creating Areas</i>	94
<i>Deleting Areas</i>	96
<i>Obtaining Information About an Area</i>	97
<i>Iterating Over All Areas Within an Address Space</i>	98
<i>Changing the Access Level of an Area</i>	99
<i>Finding the Area That Contains a Particular Logical Address</i>	99
<i>Using Areas to Access Large Backing Stores</i>	100
MEMORY CONTROL	100
<i>Obtaining Information About a Range of Logical Memory</i>	100
<i>Data to Code</i>	102
<i>Controlling Memory Cacheability</i>	103
<i>Controlling Paging Operations</i>	104
<i>Preventing Unnecessary Backing Store Activity</i>	105
MEMORY CONTROL IN ASSOCIATION WITH I/O OPERATIONS	106
<i>Preparing For I/O</i>	107
<i>Finalizing I/O</i>	113
MEMORY SHARING	114
<i>Global Areas</i>	114
<i>Client-Server Areas</i>	115
<i>Mapped Access to Other Address Spaces</i>	115
<i>Copying Data Between Address Spaces</i>	117
MEMORY RESERVATIONS	118
<i>Creating Memory Reservations</i>	119
<i>Deleting Memory Reservations</i>	120
<i>Obtaining Information About a Memory Reservation</i>	120
<i>Iterating Over All Memory Reservations Within an Address Space</i>	121

MEMORY EXCEPTIONS	122
MESSAGING.....	124
MESSAGE PORT MANAGEMENT	124
<i>Creating Message Ports.....</i>	<i>124</i>
<i>Deleting Message Ports</i>	<i>124</i>
<i>Obtaining Information About a Port.....</i>	<i>125</i>
<i>Iterating Over Message Ports</i>	<i>126</i>
MESSAGE OBJECT MANAGEMENT	126
<i>Creating Message Objects</i>	<i>127</i>
<i>Deleting Message Objects.....</i>	<i>127</i>
<i>Locking Message Objects.....</i>	<i>128</i>
<i>Unlocking Message Objects.....</i>	<i>128</i>
<i>Obtaining Information About an Object</i>	<i>128</i>
<i>Changing Information About an Object.....</i>	<i>129</i>
<i>Iterating Over Objects</i>	<i>129</i>
ABOUT MESSAGE TRANSACTIONS	130
<i>Message IDs.....</i>	<i>130</i>
<i>Message Types</i>	<i>130</i>
<i>Kernel Messages</i>	<i>131</i>
SENDING MESSAGES	132
<i>Send Options</i>	<i>132</i>
<i>Synchronous Sends.....</i>	<i>133</i>
<i>Asynchronous Sends.....</i>	<i>136</i>
RECEIVING MESSAGES	137
<i>Receive Options.....</i>	<i>138</i>
<i>Message Control Blocks.....</i>	<i>138</i>
<i>Receiving Messages Synchronously.....</i>	<i>140</i>
<i>Receiving Messages Asynchronously.....</i>	<i>140</i>
ACCEPTING MESSAGES	141
REPLYING TO MESSAGES	144
REPLYING TO A MESSAGE AND RECEIVING ANOTHER MESSAGE	146
OBTAINING INFORMATION ABOUT A MESSAGE	146
CANCELING MESSAGE REQUESTS	147
<i>Send Message Cancellation</i>	<i>148</i>
<i>Receive Message Cancellation.....</i>	<i>149</i>
<i>Client Initiated Cancellation Messages</i>	<i>149</i>
GETTING SYSTEM INFORMATION	151
RESTRICTIONS ON USING MICROKERNEL SERVICES A	
SERVICES THAT CAN BE CALLED FROM TASK LEVEL	A
SERVICES THAT CANNOT BE CALLED BY NON-PRIVILEGED TASKS	A
SERVICES THAT CAN BE CALLED FROM SECONDARY INTERRUPT HANDLERS	A
SERVICES THAT CAN BE CALLED FROM HARDWARE INTERRUPT LEVEL	B

ABOUT THE MICROKERNEL

The microkernel provides support for modern operating system features including:

- Preemptive multitasking
- Scheduler-supported synchronization primitives
- Multiple large, sparse address spaces
- Memory mapped files
- Demand paged virtual memory
- Memory protection
- Object based message system
- and, via libraries, timing, synchronization, and other services

This document consists of three parts. The first part is an overview of the major features and concepts of the microkernel, the second part is a comprehensive technical presentation of the microkernel's interfaces, and the third part is a series of appendices that cover remaining issues and compatibility.

KERNEL OBJECTS AND IDS

Most interfaces to the microkernel fall in to one of three categories; those that create something, those that manipulate something previously created, and those that delete something. The things that are created, manipulated, and deleted are known as *kernel objects*.

Kernel objects include:

- Address spaces
- Address areas
- Memory backing objects
- Tasks
- Kernel processes
- Timers
- Event groups
- Software interrupts
- Message objects
- Message ports
- Messages

This document describes how to create kernel objects of various types. It describes their properties and behaviors. It discusses how to manipulate and destroy kernel objects and describes when they are destroyed as the side effect of some other operation.

You cannot directly manipulate kernel objects because the underlying data structures are not a part of the application programming interface (API) to the kernel. In certain implementations, the objects themselves may not be directly addressable to software other than the microkernel.

When the microkernel creates a kernel object, it generates an identifier (ID) for that object. IDs are 32-bit values that uniquely identify a particular object. Functions that create an object return the ID of the newly created object. Functions that act upon or destroy an object require that you pass the ID of the kernel object that is to be acted upon or destroyed.

IDs are completely opaque. The techniques used to associate an ID with the underlying object are private to the microkernel. IDs cannot be used to access the underlying data structures. The actual memory used to store kernel objects is not necessarily in the same address space as clients of the microkernel.

Because the microkernel does not support persistent objects (objects that survive across system boots), it has no need for persistent IDs. IDs are unique only for the duration of a particular boot. Additionally, IDs are unique only to a given kind of

kernel object. That is to say there is a separate flat ID space for each kind of kernel object. If you create two separate kinds of kernel objects (such as a task and a message port), it is possible that the same ID value will be returned for each of them. It is the responsibility of the programmer to ensure that the ID of a particular kind of kernel object is used only in conjunction with operations on that kind of object. If you perform message operations on task IDs, there is a slight chance that, because the task ID is also a valid message ID, undesirable side effects may result.

Using an ID after the underlying object has been implicitly or explicitly deleted is erroneous. Typically, the microkernel detects such usage and returns an error. However, IDs are subject to reuse when the kernel object to which the ID was originally assigned is reclaimed. Every effort is made to minimize the amount of reuse to assist in the detection of programming errors and to improve system robustness.

Because only the creator of a kernel object has its ID, the creator can limit access to the object by controlling access to the ID, thereby providing a measure of security. The mechanisms for generating and decoding IDs are private. Making these mechanisms public would compromise the currently available level of security and would prevent the implementation of additional levels of security in the future.

NAMING

The microkernel's consistent use of IDs is motivated by the desire to isolate the underlying data structures for reasons of both robustness and security. One problem with this approach concerns entities that must be well known throughout the system. Historically, solutions to this problem have one of two forms: place the IDs of these entities in well known locations (such as low memory) or provide a service whereby the IDs can be found through some naming conventions.

The microkernel uses a name-based registry into which well-known IDs are placed at the time they are created. The registry supports operations to create, delete, and lookup entries.

EXECUTION - TASKING AND INTERRUPTS

The tasking and interrupt mechanisms of the microkernel formalize the environments for execution of software by the processor. This section provides an overview of these concepts.

About Execution

A significant amount of the microkernel design is devoted to the manner in which code is executed. Considerable effort has been spent to ensure that high-level language software can be used directly without interfacing glue and to normalize the execution environment for applications, VBL tasks, Time Manager tasks, and I/O completion routines, which all execute under different rules in System 7. This part of the design is largely intangible in that there is little or no implementation part behind the design. Mostly, the design details the environments in which execution happens. These environments include:

- Task level - This level is where nearly all code is executed. Application programmers typically are only concerned with task level execution. The processor is executing at task level whenever it is not processing interrupt level code.
- Hardware interrupt level - This level is usually of concern only to driver writers and certain internal OS software developers. Hardware interrupt level execution happens as a direct result of a hardware interrupt request.
- Secondary interrupt level - This level is similar to the deferred task concept in System 7. The secondary interrupt queue is filled with requests to execute subroutines that are posted for execution by hardware interrupt handlers that need to perform certain actions but chose to defer the execution of those actions in the interests of minimizing interrupt level execution. Unlike hardware interrupt handlers that can nest, the execution of secondary interrupt handlers is always serialized. For synchronization purposes, task level execution may also post secondary interrupt handlers for execution; these are processed synchronously from the perspective of task level, but are serialized with all other secondary interrupt handlers.
- Kernel level - The rules and guidelines for executing certain portions of the microkernel are different from the rules and

guidelines for any exported environment. The microkernel's environment execution environment is private and is not described in this document.

Each of these execution environments has common attributes. For example, whenever any software executes at task level, it uses the stack created for that task at the time the task was created.

Different execution levels have different restrictions. Task level execution can make use of nearly any microkernel, OS, or ToolBox service. Secondary interrupt and hardware interrupt handlers are allowed only a subset of those services. Furthermore, only task level execution is allowed to access memory that is not physically resident; page faults at either hardware interrupt level or secondary interrupt level are illegal and system fatal.

About Tasks

The primary unit of execution within the microkernel is called a *task*. This term is frequently interchanged with the term *thread* in other operating system and kernel architectures.

Tasks are used to virtualize the existence of the physical processor and provide the illusion of many processors, each performing a different kind of work at the same time. In a microkernel-based system, a separate task exists for each application. Additionally, applications are free to create additional tasks if it is desirable to do so. The microkernel I/O system is also based upon tasks with a separate task potentially used for each device driver.

The processing resources available to a task are called the task's *context*. Context includes general purpose registers and special purpose registers. Note that task context is processor dependent.

Along with processor context, a task requires the presence of certain other resources. These include the task control block and the task stacks. The task control block is an internal data structure that describes the task to the microkernel; it is only accessible to the microkernel and is always referred to by a task ID. In addition, each task has at least one stack.

The process of ceasing the execution of one task and beginning the execution of a different task involves saving the context of the former task and restoring the context of the latter task. This combination of a context save and a context restore is called a *context switch*.

The mechanics of context switching are relatively simple. However, the decision of when to context switch and which tasks to context switch, known as *scheduling*, is rather complex. Scheduling logic is a key differentiating factor between different operating system and kernel architectures.

About Task Scheduling

The microkernel uses an event-driven, priority-based, preemptive scheduler.

Event-driven means that scheduling decisions are made coincidentally with certain key events that occur within the system. Interrupts are one example of an event that drives the scheduling process. Other examples include setting or waiting for an event flag, notifying or waiting on an event queue, and sending or waiting for a message. Note that these scheduling events are different from the OS, ToolBox, and EPPC events that drive applications.

Priority-based scheduling implies that each task's importance is used when selecting a task for execution. A task's relative importance is specified by its *priority*. Microkernel tasks have a priority between 1 and 30; the larger the value the higher the priority. When a task is created, it is given a priority that can be increased or decreased at any time.

A task is eligible for execution whenever it is not waiting for an operation to complete. These waits can be explicit, as in the case of synchronous I/O operations, or implicit, as in the case of page faults. Tasks that are not eligible for execution are said to be *blocked* upon some event. Many tasks may be eligible for execution but only one can be executing at any instance. Under the microkernel, the task with the highest priority that is eligible for execution is guaranteed to be the task that is executing.

Preemptive scheduling means that the system, not the current executing task, controls scheduling. In System 7, the scheduling of applications is purely cooperative and the resultant system requires well-behaved applications if it is to function in a way that is pleasing to the user. If an application fails to cooperate, it can interfere with the operation of the entire system. Preemptive scheduling alleviates most of the need for cooperation. When the event upon which a task is blocked occurs, that task is again made eligible for execution. If that task has a priority greater than the currently executing task, a context switch is performed and the higher priority task immediately resumes execution from the point at which it was blocked.

The microkernel scheduler uses time-slice scheduling to manage tasks that have equal priority. If several tasks are eligible for execution at the highest priority, each is allowed to execute for an internally-specified period of time called a *time-slice*. When its time slice has expired, the context of the currently executing task is saved and the context of the next task at that same priority is restored. In this way, each task at the highest priority is given access to the CPU in a round-robin fashion. No single task can prevent other tasks from executing unless it is the only task at the highest priority.

Time-slicing is only used when several tasks are all eligible for execution at the same priority and no higher priority tasks are eligible, so time slicing never interferes with the priority-based scheduling algorithms. If a higher priority task becomes eligible for execution, it always gets immediate access to the CPU.

Applications should not use the microkernel's scheduling behavior as a substitute for explicit synchronization. Under System 7, it is common practice to depend upon the scheduling behavior of the system (that is, by not calling `WaitNextEvent`) to synchronize access to shared data structures. Under the microkernel, applications should always use explicit synchronization mechanisms instead of making assumptions about which tasks can run based on relative priorities. For example, a higher priority task might page fault, thus allowing a lower priority task to execute.

The microkernel does not include specific support for real-time scheduling. You cannot specify that a task is to execute next or that a task should execute at a certain time or that a task should receive a certain percentage of CPU time. The microkernel scheduler does not contain support for deadline scheduling.

Currently, a task's priority is not adjusted implicitly by the kernel as is done by the Windows NT® Boost/Decay scheduling policy. Such support may be added at a later time. Most microkernel clients should not specify specific priority numbers for the tasks they create. Instead, they should specify priorities using the provided categories, which include application, server, and background.

About Software Interrupts

In addition to the scheduling of multiple tasks on a single processor, the microkernel scheduler provides a mechanism for executing asynchronous completion routines.

In System 7, an I/O completion routine associated with an asynchronous I/O request is usually run at interrupt level and is completely asynchronous to the

execution of the application that started the I/O. This means that the completion routine runs in a completely different environment than the initiator of the request. The completion routine gets parameters in registers rather than on the stack. The completion routine cannot access static variables or the jump table because A5 is not setup by the system. Finally, the invocation of the completion routine is in no way related to the importance of the requester. Because the invocation happens at hardware interrupt time, application code is invoked in a completely uncontrolled way.

The software interrupt feature of the microkernel scheduler allows a specified subroutine, with specified parameters, to be executed within the context of a given task, but asynchronously to that task's otherwise normal execution. A microkernel-based system uses software interrupts to implement many features that are implemented with hardware interrupt handlers in System 7, such as VBLs, Timers, I/O completion routines, etc.

Within a given task context, software interrupts are processed on a first-in, first-out basis; they do not nest. When a software interrupt handler finishes and no other software interrupts to that task are pending, the task simply resumes execution at the point prior to the software interruption. A given task can enable and disable its ability to receive software interrupts and interrupts are queued to the task until they can be delivered. Software interrupts do not affect the scheduling policies of a given task with respect to other tasks.

Any task can send a software interrupt to any other task. The microkernel also uses the software interrupt mechanism to inform clients that a request has completed.

About Privileged Execution

Most software in a microkernel-based system is non-privileged. Non-privileged software executes with the CPU in user mode. All applications run in user mode. Some kinds of software (such as, device drivers) are best executed in supervisor mode and are, therefore, privileged. Privileged software has full access to the machine's instruction set and memory addressing capabilities. Execution mode is a task attribute; when you create a task, you specify whether it is to be a privileged or a non-privileged task.

Privileged tasks always execute in supervisor mode. They have a single stack for all execution and local variable storage.

Non-privileged tasks can execute in both modes. Most of a non-privileged task executes in user mode. Because the microkernel always runs in supervisor mode, when a user mode task calls the microkernel, the microkernel's execution takes place in supervisor mode. Therefore, non-privileged tasks have two stacks; one for user mode execution and one for supervisor mode execution.

About Synchronization

The preemptive nature of task scheduling requires explicit attention to task synchronization. Synchronization of accesses to shared memory or I/O devices is frequently the most difficult aspect of programming in a multi-tasking environment. The microkernel provides event groups and event queues to allow the synchronization of tasks around critical sections. Event groups and event queues are discussed in the section "Synchronization Mechanisms," later in this document.

About Interrupts

Interrupt handlers are subroutines that are invoked by the microkernel in response to a particular hardware interrupt request. Interrupt handlers execute in supervisor mode and have access to a single interrupt stack. The possibility of nested interrupts can cause several interrupt handlers to each be activated on the interrupt stack simultaneously.

Interrupt handlers are formally registered with the microkernel. You do not install them directly into a vector table. Only a single handler may be registered for any given interrupt source, and you cannot install a handler without first removing the previously installed handler.

Interrupt sources are designated by a hardware-dependent vector number. This number is not related to the processor architecture's vectoring scheme. It is a simple enumeration of the interrupt sources.

The design philosophy for the microkernel's interrupt system is driven by the desire to minimize interrupt latency and, therefore, maximize responsiveness. This enables better real-time response and also allows greater I/O throughput.

The microkernel provides a mechanism for performing real-time processing, in response to interruptions, outside of interrupt level. This mechanism is called the *secondary interrupt handler*. Secondary interrupt handlers are similar to deferred tasks in System 7. Secondary interrupt handlers are queued by hardware or *primary* interrupt handlers. When you queue a secondary interrupt handler, you

specify the handler and a set of parameters with which it is to be invoked. The handler is not called immediately. Instead, the information is placed into the *secondary interrupt queue*.

In order to synchronize with interrupt-level execution without disabling hardware interrupts, task-level software can also insert subroutines in the secondary interrupt handling queue. The queue is always processed first-in/first-out, and the execution of the queued handlers is always serialized. Although hardware interrupts remain enabled and hardware interrupt handlers can preempt secondary interrupt handlers, secondary interrupt handlers cannot preempt one another.

The secondary interrupt handler queue is always emptied prior to running any task level software.

When writing device drivers that handle hardware interrupts, it is important to balance the amount of processing done within your primary and secondary interrupt handlers along with that done by your driver's task. You should make every effort to push processing time out of primary interrupt level into secondary interrupt level and, similarly, push secondary interrupt level processing into your driver's task. Doing this allows the system to be tuned so that your driver's processing time is balanced with the needs of other drivers and applications.

KERNEL PROCESSES

During the execution lifetime of any software, that software allocates and deallocates many kernel objects. When that software terminates, either normally or abnormally, the system reclaims any of the kernel objects that were not deallocated.

The locus of resource allocation and reclamation within the microkernel is called a *kernel process*. This term is equivalent to the term *process* in other operating system and kernel architectures. The term *kernel process* was chosen because the term *process* is already used and well understood in System 7 Process Manager nomenclature.

A kernel process is composed of tasks and other microkernel resources. The execution of those tasks may create and destroy additional microkernel resources (including other tasks) during their lifetime. Each of these resources is said to belong to the kernel process. Kernel processes are completely passive. Kernel processes do not execute instructions; tasks execute instructions and tasks belong to a specific kernel process.

Kernel processes also designate a set of memory locations and associated values, known as an *address space*. Each kernel process has access to exactly one address space. However, a single address space may be shared by several kernel processes. A given task, belonging to a given kernel process, executes within the address space of that kernel process. The task can only access memory locations associated with its kernel process' address space. In this way, kernel processes provide not only resource reclamation but also memory protection.

The creation of a kernel process returns a KernelProcessID. All subsequent operations upon the kernel process require that the kernel process be specified by ID. When the kernel process is reclaimed, all resources that belong to the kernel process are also reclaimed. Kernel processes are reclaimed either explicitly (possibly by their creator) or implicitly when all tasks with the kernel process have terminated. In this way, problems of garbage generation are handled in a well-controlled and easily understood manner.

The microkernel and all the privileged tasks in the system are included in a single kernel process.

ADDRESS SPACE MANAGEMENT

Addressing is such a basic concept in computer systems that it is frequently taken entirely for granted. However, as operating system and application software grow in complexity, the manner in which memory is utilized becomes increasingly important.

A significant portion of the microkernel is devoted to implementing a rich set of addressing and memory management mechanisms. These mechanisms provide the foundation for many of the high-level features desired in Macintosh systems: memory protection, memory mapped files, and high performance virtual memory.

About Addressing

Because so much about addressing is taken for granted, a brief overview of terminology and concepts follows.

An *address space* is the domain of addresses that can be directly referenced by the processor at any given moment. A *logical address* specifies a location within an address space. Logical addresses are unsigned in nature; the lower bound of a logical address is zero, and the upper bound is the size of the address space minus one. For example, in a 4 GB address space there are 2^{32} (4 GB) distinct logical addresses for bytes, ranging from zero to $2^{32} - 1$. The number of bits required to represent logical addresses (the size of the address) is often used to denote the size of the address space. For example, a 4 GB address space can also be called a 32-bit address space.

Some systems provide a single address space that is in effect for all software. Others provide distinct address spaces for different software entities. This so-called *multiple address space* model provides isolation of software and restricted access to hardware (known as *protection*). When combined with the ability to use secondary storage, usually hard disk, as an extension to a computer's physical memory (a technique called *virtual memory*), the resultant memory model offers many advantages. These include the ability to address large amounts of protected memory at a cost that is quite low in terms of performance and dollars.

There is an association between logical addresses and hardware. When the processor references a given logical address, there is an effect on the hardware. Usually, the hardware is RAM, and the effect is to acquire or modify data in that RAM. Alternatively, the hardware may be a device that provides some auxiliary

function (such as network access), and the effect is to control that device's operation.

In simple memory models, the association of logical addresses with hardware is statically determined by how the hardware is wired to the processor. In the relatively more complex models implemented by virtual memory systems, the association is made dynamically. Further, the association can be extended to hardware not directly accessible by the processor, such as secondary storage. Forming an association for a range of logical addresses is called *mapping*. A range that has an association is said to be *mapped*, and a range that does not have an association is said to be *not mapped*.

Virtual memory systems require specialized hardware support for mapping. For architectural reasons, this support hardware always forms mappings based on address ranges rather than on a per-address basis. These ranges become the unit of mapping. If mapping ranges are fixed-sized, the mapping units are called *pages* or *logical pages*. To provide address spaces with sizes greater than the amount of RAM, the virtual memory system uses the support hardware to shuffle RAM among different logical pages. Doing this on an as-needed basis is known as *demand-paging*.

A reference to a logical address that is mapped to secondary storage, but whose data is not immediately available in RAM, is referred to as a *page fault*. In response to a page fault, the microkernel initiates the appropriate transactions to obtain the contents of the logical page and then maps that page into the address space. With the fault repaired, the microkernel causes the execution of the faulting software to resume at the point of the fault. The entire effect of the page fault is transparent to the software that caused the fault.

The System 7 Addressing Model

System 7 provides an addressing model that is molded around a single, completely open address space that provides no protection of software or hardware. Originally, this space was shared by the system and one application; now it is shared by the system and multiple applications. The Virtual Memory introduced by System 7 did not change this. For compatibility reasons, Virtual Memory was not allowed to provide separate address spaces or protection, and was forced to settle for extending the existing single address space by about a factor of two. Virtual Memory's purpose is confined to preventing users from having to buy more RAM. Internally, its method for RAM management complicates the model for non-application software and introduces substantial address space overhead.

Logical addresses may be either 24- or 32-bits, on a per-system boot basis. This coexistence complicates both the internal workings of the system software and the implementation of third-party software, but has been considered necessary for backward compatibility.

The simultaneous execution of multiple applications within a single, limited address space means contention for memory among those applications. The Temporary Memory scheme was introduced by MultiFinder (an earlier version of the Process Manager) to provide an outlet for applications that need "emergency" dynamic memory allocations. This is possible because, often, there is more memory where MultiFinder gets it. Temporary Memory is really just a stop-gap measure because applications have no other way to fully utilize the address space.

The Microkernel Addressing Model

The microkernel's addressing model is designed with modern hardware and software architecture in mind.

Multiple Address Spaces

The microkernel provides operations to create and destroy address spaces. The contents of a newly created address space are based upon a template maintained by the microkernel. This template causes slot space, frame buffers, ROM, and the microkernel to be mapped into each address space with appropriate access protection. The remainder of the address space is devoid of content.

Areas

A range of logical address space that is mapped is called an *area*. Areas begin and end on page boundaries. Operations are provided to create and destroy areas. The area creator must specify the size of the area, how the memory content of the area is to be derived and maintained and what access rights are available to various clients.

Areas can be derived from disk files or their initial contents may be unspecified. In either case, the memory management system provides support for clearing the contents of the area on a per-page basis when the page is first accessed. Areas can be maintained in a manner that causes each logical page within the area always to be physically resident or to be paged in and out of physical memory as needed. Area attributes govern the ability of privileged and non-privileged

execution to read and write the area and for the area to be shared among multiple address spaces.

When an area is created, it can be surrounded by *guard pages*. These guard pages are excluded pages in the address space that assist with detecting accesses beyond the area. Guard pages can also be used to detect stack overflow.

Paging

The sum of the pages in all areas in all address spaces typically exceeds the amount of physical memory. This shortfall is made up by using secondary storage, known as *backing storage*, to store data which cannot be physically resident.

The microkernel is responsible for the movement of data between backing store and memory. The page replacement policies utilized by the microkernel attempt to minimize the frequency of page faults by retaining the most recently used pages in physical memory and allowing infrequently used pages to migrate to backing storage.

Paging performance is further enhanced because the software involved in resolving page faults is limited to the microkernel itself and those drivers involved in accessing the paging device. Therefore, only a small number of logical pages must be held in physical memory allowing a much greater number of physical pages to be used for frequently accessed data.

All I/O performed by the microkernel to satisfy page faults is performed through *backing objects*. Backing objects are message objects that respond to messages specified by the microkernel and perform the appropriate I/O operations. Backing objects isolate the microkernel from the mechanics of finding the appropriate data on the storage device. Page faults can therefore be satisfied from nearly any I/O device including networks, hard disks, tape, etc.

Global Areas

Sharing certain resources, especially code, among many clients is an important concept in modern software. The microkernel provides effective, efficient support for sharing code and data among clients in separate address spaces.

When creating an area, the `kGlobalArea` option causes the contents of the area to be addressable in every address space. The contents disappear from all address spaces when the area is destroyed. Global areas have the protection attributes

specified at the time of their creation regardless of the address space from which a reference is performed.

Because global areas are visible to all address spaces, space in every address space must be available to create any global area. The microkernel sets aside a predetermined amount of space in every address space for use by global areas. Once exhausted, no additional global areas can be created until others have been deleted. The amount of address space set aside for global areas is unspecified.

All code executed in privileged mode, and all data directly referenced by privileged tasks and interrupt handlers must be in globally shared areas. Microkernel services are available to allow privileged code to access non-globally-shared data.

I/O Coordination

When I/O operations are performed between an external device and memory, several aspects of the memory's contents must be coordinated. Typically, the logical contents must be made physically resident so they can be accessed at hardware interrupt or secondary interrupt level where page faults are not allowed. Additionally, the coherency of any data and instruction caches must be maintained to ensure that the data being moved is not stale and that the effects of the data movement are observed by the processor.

When using DMA hardware to perform the I/O operation, it is also necessary to translate the logical address range into a set of physical address ranges. This set of physical address ranges is called a *scatter-gather list*.

The microkernel provides efficient support to prepare a range of addresses for an I/O operation and to clean up that same range when the operation is finished. Through the use of appropriate parameters, all cache manipulations are performed, the data is made physically resident, and a scatter gather list is generated. The client need not be concerned with the cache topology or any other aspect of the hardware because the microkernel provides complete isolation.

Addressing And Execution

The relationship of execution to addressing is at the kernel process level. When a kernel process is created an address space must be designated for that kernel process. Any tasks created within that kernel process will see this logical address space. Several kernel processes may all share a single address space, but they will not be protected from each other. The kernel process that contains the microkernel and all the privileged tasks in the system has a special addressing

environment. When any task in this kernel process is executing, only global areas are addressable.

Inter-Address Space Access

During normal execution, a given task has access to only the memory that is mapped into its kernel process's address space. It is possible, however, to gain access to the logical memory of other address spaces. The general mechanism for shared memory is to map the same backing store data into the various clients' address spaces. Further routines enable straightforward data copying and cross-address space mapping. An additional facility is provided to arbitrate sharing memory at the same location in each address space.

Code Fragment Manager Contexts

The Code Fragment Manager supports the notion of a *context* into which code is loaded. Most programs and libraries are specified to have *per-context* global data — each time the code is referenced from a new context, a new copy of its global variables is created. The Code Fragment Manager associates a context with each kernel process. Since all privileged software executes within the single kernel process, a privileged context is shared by all privileged code.

SYNCHRONIZATION MECHANISMS

There are two microkernel-supported mechanisms for synchronization of task level execution: *queues* and *event groups*. A queue is a serialized first-in/first-out list of elements. An event group is a set of 32 flags or binary semaphores that can be acted upon individually or in combination.

Note: In addition to these microkernel services, processor-specific synchronization primitives are available, such as the *lwarx/stcx* instructions on PowerPC processors. These capabilities are combined with event groups in a Synchronization Services library that provides lightweight locks, read/write locks, and atomic operation functions. Refer to the Synchronization Services ERS for details.

Event Groups

Event groups are created explicitly. There is no limit on the number of event groups that can be active in the system at one time. Each event group is referenced by an ID and contains 32 unique *event flags*.

Once created, any task can operate on a given event group. Operations on the group manipulate one or more of the group's event flags. The operations are read, set, clear, and wait.

Reading an event group returns the value of the 32 event flags. This operation has no side effects on the task that is reading the flags or on any other task.

Clearing event flags is done by specifying an event group and a 32-bit mask. Each flag that is set in the mask is cleared in the event group. This operation does not effect the clearing task or any other tasks in the system.

Setting event flags is done by specifying an event group and a 32-bit mask. Each flag that is set in the mask is set in the event group. This operation may cause other tasks that are waiting on the event group to become executable.

Waiting for event flags is done by specifying the group, a mask of flags to wait for, and a waiting operation. The mask contains 32 bits and indicates, in conjunction with the operation, a condition for which the calling task wishes to wait. The operation specifies whether the condition is satisfied by *any* of the events in the mask becoming set or only when *all* of the events in the mask become set. Additionally, the operation indicates if the events, specified by the mask, are to be cleared when the condition is satisfied.

If all waiting tasks specify the clear option, exactly one task will be unblocked each time the condition is satisfied. If no tasks specify this option, all waiting

tasks will be unblocked. Waiting tasks are queued in priority order, so that the highest priority waiter is unblocked first when its condition is satisfied. Within a particular priority, tasks are queued on a first-in/ first-out basis.

Wait operations can include a time limit that specifies the time the calling task is willing to wait for the specified condition to occur. If the time limit is exceeded, the task is made executable even though the condition has not been satisfied.

Event groups may be used to implement many styles of synchronization mechanisms. The ability to wait upon a combination of events is especially useful in avoiding many deadlock situations that arise with binary semaphores.

Queues

Queues are similar to event groups in many ways. There is no limit on the number of queues that may be active in the system at one time. Each queue is explicitly created and is referenced by an ID.

Once a queue is created, any task can operate on a given queue. The two queue operations are notify and wait.

When a queue is notified, it is given two pieces of information. The first piece of information is a user-specifiable refcon. The refcon can be any data that the notifying task wishes to provide to the receiving task. The second piece of information is the actual queue element. The queue element includes both the type of data and the data itself.

If a task is already waiting on the queue at the time of notification, the queue data is delivered to the task and the task is made executable. If there are no tasks waiting on the queue, the data is enqueued for the next task that waits on the queue.

Queues can be waited on by specifying the ID of the queue, the queue data that the task is interested in, and a time limit to wait. If data is already queued, the queued data is returned to the waiter. Otherwise, the task waits on the queue, relinquishing its execution time to other tasks in the system. Like event groups, if the time limit is exceeded, the task is made executable without any queue data being returned.

It is possible to use queues for a wide variety of synchronization purposes. Multiple tasks can wait on the same queue and process requests concurrently. The microkernel also provides queue notification for many of its asynchronous events, including messaging and task termination.

MESSAGING

As system software becomes more modular, the flow of information between modules becomes critical for robustness and performance. Preemption and multiple address spaces increase the requirement for smooth communications between modules. Problems of synchronization must be overcome and the ability to communicate across address space boundaries is required. The microkernel message system supports communication across address space boundaries and provides synchronization between modules.

Messages

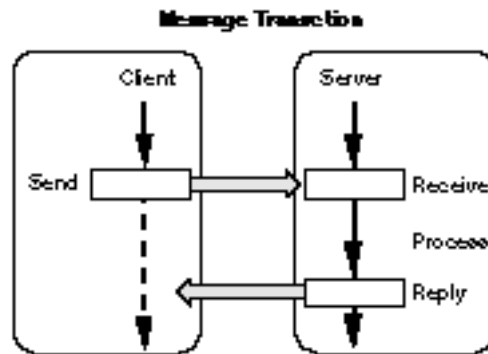
A message is a unit of information interchange. The microkernel is not concerned with message content; it neither examines nor interprets the content of the messages. Rather, the microkernel assists in moving the message from the originator to the recipient, while providing the ability to control and prioritize the flow of information. The message system is suited for the exchange of control and status information as well as for the exchange of data.

Client-Server

The microkernel message system uses a *client-server* model of communications. In this model, a service is provided by a server. Software that wishes to make use of a service is called a client of the service. The message system allows data to be transported from a client to a server and for the server to notify the client of the results.

Transactions

When a client makes a request of a server it does so by *sending* a message. The server must actively participate by attempting to *receive* messages from its clients. When a server has received a message, it performs the implied work and notifies the client by *replying* to the message. This combination of send, receive, and reply is called a *message transaction*. The microkernel provides all transaction support, including any synchronization and address space mapping operations that may be required. The following figure illustrates these concepts:



Moving Data

The data that flows between the client and server usually conforms to semantics specified by the server. Data flows from client to server and from server to client. All data is described to the microkernel through the use of address/byte count pairs. The data that flows from the client to the server is called the *message contents*. The data that flows from the server back to the client is called the *reply data*.

The message contents and reply data address/byte count pairs are conveyed to the server at the time the server receives the message. If the message is sent across address space boundaries, the microkernel can, at the client's discretion, map the message contents directly into the server's address space or copy the contents into the server's address space. The microkernel can also choose to map the reply data buffer into the server's address space.

Data is buffered in the microkernel only if the `kSendIsBuffered` option is specified. Therefore, the client must not attempt to modify or deallocate either the message contents or reply data buffers until the transaction is complete.

Some messages, specified as part of the send operation, include the addresses of other data that are associated with the message. This technique is commonly used for reading data. Read requests typically indicate the source of the data (such as a file offset) and specify the address of a client buffer into which the server should place the data. These regions of memory that are associated with the message by both the client and the server are unknown to the microkernel. It is the responsibility of the server to ensure that it can address them. Microkernel services are available to perform the mapping or copy operations needed to implement such services.

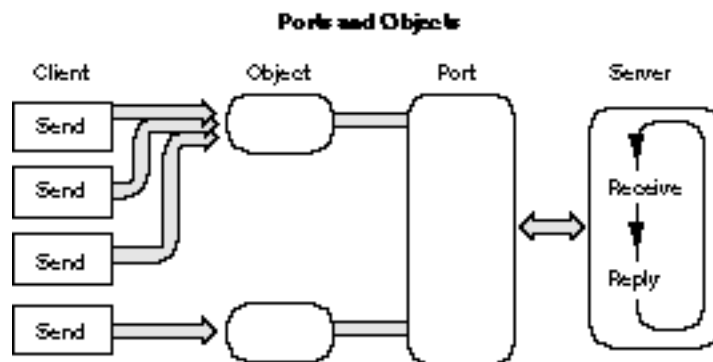
Ports and Objects

Message objects are abstract entities that represent various resources to message system clients. These objects may represent resources, such as devices, files, and windows. Clients send messages to *objects*.

Message ports are abstract entities that represent a service. These ports may represent a device driver, a file system, or a window manager. Servers receive messages from *ports*.

Objects are said to belong to a port. A message sent to an object is received from that object's port. The client is usually unaware of the port associated with a particular object.

The duality of objects and ports allows efficient support in situations where a number of separate entities, all conceptually different from the client's perspective, are served by a single server and with identical actions.



Ports and objects are created by the message system on behalf of a server. The creation of an object requires designating a port from which messages sent to the object will be received. Therefore, ports must be created prior to their objects. Once created, an object may be migrated from one port to another. This allows servers to control port utilization for whatever reasons they choose. For example, objects that are highly utilized can be migrated to a port that is served by several tasks within the server.

Objects contain a single 32-bit value, specified at creation time, that is used by the server to identify the object. This value, called a *refcon*, allows the server to associate any per-object information with the message. When receiving messages from a port, the server is provided with the message as well as the refcon from the object to which the message was sent. The server can use refcons for any purpose; they are not examined or interpreted by the microkernel. Typically, the

refcon is the address of a control block for the object; a file object's refcon could be the address of the file control block for that file. The server can examine and change the refcon of an object at any time.

Ports and objects are referenced by IDs.

Sending Messages

The process of sending a message can be either synchronous or asynchronous. The synchronous send operation blocks the client until the server acts on and replies to the message. The asynchronous send operation allows the client to continue execution while the server processes the message.

Synchronous send operations return a status value that indicates the success or failure of the message transaction. Errors may be returned by either the microkernel or by the server.

When a message is sent synchronously, the sender can specify a time limit. The value of the time limit controls how long the sender is willing to wait for the transaction to complete. Should the time limit be exceeded, the message is canceled by the microkernel.

Asynchronous send operations yield two separate status results: a send status that is returned when the asynchronous send call returns to the sender and a reply status that is delivered asynchronously to the client when the server finishes processing the request.

The client receives notification that the server has finished processing an asynchronously sent message in any or all of three different ways. First, the client can specify a memory location that is to be updated with the 32-bit message reply status. Second, the client can specify an event group and set of flags within that group that should be set. Finally, the client can specify a software interrupt that should be delivered.

Receiving Messages

Servers receive messages from ports. Servers can receive messages in three separate ways: synchronous receives, asynchronous receives, and acceptance functions. All three methods of receiving messages require that the server explicitly designate a port from which the messages are to be taken.

Synchronous receive operations block the execution of the server until a message arrives at the port. The server may limit the length of time that the server remains blocked waiting for messages. If the time limit is exceeded, the server again begins to execute and is informed of the time-out.

Asynchronous receive operations do not block the execution of the server. Instead of blocking, the server continues to execute and is notified when the next message arrives at the port. The server can request that the notification be delivered in any or all of three ways: a memory location update, an event flag update, or a software interrupt delivery. These notifications are the same as the notifications described in the previous section for asynchronous send operations.

The third method of receiving messages is by registering an acceptance function. Acceptance functions are simply subroutines that are called in-line in the context of the sender at the time the message is sent. Acceptance functions are always called in supervisor mode and, therefore, not all servers can register them.

Numerous synchronous and asynchronous receives can be made of a single port, but only one acceptance function can be registered. When a message is sent, it is given to only one receiver. The process of matching a sent message to a receiver is governed by message type. Message types are described later in this document.

Regardless of the manner in which a message is received (synchronously, asynchronously, or acceptance), the server is provided with more than just the message. The refcon of the object to which the message was sent and an ID for the message, are also returned to the server. The refcon allows the server to associate information about the object with the message. The message ID is used by the server to notify the client that processing of the message is complete.

Replying to Messages

When a server finishes processing a message, it must inform its client. The process of notifying the client is called a *reply*. When a server replies to a message, the reply includes the message ID of the processed message and a 32-bit status. The microkernel does not interpret the status in any way. Rather, the status is interpreted by the client in a way that is defined by the interface between client and server.

Servers must reply to all messages they receive. Synchronous senders remain blocked until the server replies. Servers can implement time limits upon their transactions to prevent the system from becoming deadlocked.

Message Types

All message send operations require the specification of a 32-bit message type. When a server makes a receive request, it also specifies a 32-bit message type. The message system does not interpret the message type, but it does use it to match senders with receivers.

The message system matches a receiver with a message by ANDing the message type specified by the sender with the message type specified by the receiver. If the result of the AND is non-zero, the message is given to that receiver. When scanning the receivers looking for compatible message types, the message system checks acceptance functions first. If no acceptance function matches the message type, the message system checks the synchronous and asynchronous receivers. If no receiver can be matched with the message, the message remains in the port until a receive operation is performed that matches the message type.

A receive operation that specifies a message type value of 0xFFFFFFFF receives all messages, regardless of type. This includes messages sent with a type value of zero.

One bit of the message type is used by the microkernel for certain special system-generated messages. Messages of this type are defined by Apple and should be supported by your server. Clients should, however, refrain from doing so. The microkernel message type and the pre-defined messages are covered in the detailed description of the message services later in this document.

Canceling Asynchronous Message Operations

When using asynchronous services, it is occasionally desirable to withdraw operations that have been started but have not yet completed. The act of withdrawing these requests is called *canceling* the outstanding request.

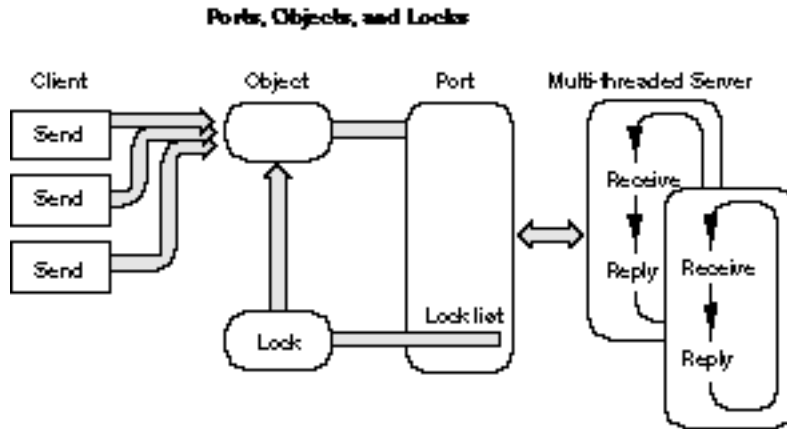
The asynchronous send and receive services each return a transaction ID that remains valid until the request is satisfied. These IDs may be used, when appropriate, to cancel the pending send or receive request.

Cancellation of asynchronous send requests is handled in one of two ways. If the send has not yet been matched with a receive request from the server, the send is simply withdrawn and the server is not affected in any way. If, on the other hand, the server has already received the request, the server is sent a special message (designated by use of the microkernel message type) that indicates that the client wants the request canceled. This special message includes the ID of the transaction so that the server knows which request is being canceled.

Canceling asynchronous receive requests simply removes the pending receive from the message port. These operations have no side affects.

Locking Message Objects

At times, a server may want to prevent messages from being sent through an object and arriving at the object's port. This is especially true when the server needs synchronize object modifications with message receipt. For this purpose, the microkernel provides services that lock and unlock message objects.



Message objects are said to be in one of three states: unlocked, locking, and locked. A message object is unlocked until an attempt is made to lock the object by using the LockObject service. At this time the message object enters the locking state. While in the locking state, messages sent to an object do not reach that object's port. Instead they pile up at the object and are not eligible to be received by the port's server. Messages that had been sent through the object to the port but had not yet been received by the port's server are removed from the port and placed back at the object. These messages are similarly not eligible to be received by the port's server. Messages that have been sent through the object to the port and have been received by the port's server prior to the lock request are not affected in any way.

An option to the LockObject service allows the caller to specify that the locking-to-locked transition should occur with either zero or one received but unreplied messages. The caller of the LockObject service is blocked until this condition is reached. Once the condition is reached, the task is unblocked and the message object is said to be locked.

In the locked state, newly sent messages continue to pile up at the message object. They are not eligible to be received. The task that made the LockObject service request should perform whatever actions are appropriate and then either

unlock or delete the locked object. Until the object is unlocked or deleted, clients of the object could be waiting for messages to be processed through the object.

While in the locking or locked state, cancel requests for messages sent to the object are processed normally. This means that the cancel requests are placed in the object's port.

Once a locked object is unlocked, any messages that were sent while the object was locked pass through the object and arrive at the object's port and can be received. These messages contain the refcon value of the object at the time the UnlockObject service was called.

Only one client can lock a given object at any time. If a request is made to lock an object that is either locking or locked, that request is blocked until the object becomes unlocked.

TIMING & TIMERS

The timing services enable the precise measurement of elapsed time. The timer services of the microkernel allow tasks to suspend their execution until a given time or to cause a specified subroutine to be called at a given time.

Measuring Elapsed Time

Measurement of elapsed time is done by obtaining the time before and after the event to be timed. The difference of these two values indicates the elapsed time of the event. In this context, time refers to the 64-bit AbsoluteTime count that is maintained by the microkernel. The count is set to zero by the microkernel during its initialization at system startup time. Conversion routines are provided in a shared library to convert from AbsoluteTime to 64-bit Nanoseconds or 32-bit Durations.

Suspending Execution

A given task can suspend its execution until a specified time in the future. This process is called *delaying*. When this time is reached, the task again becomes eligible for execution. The task does not actually execute until it is scheduled for execution according to its priority and the priorities of the other eligible tasks. In any case, the task never executes prior to the specified time.

When a task uses a delay service, it can specify the time, in relative or absolute terms, at which it should resume execution. Relative times indicate that execution should resume, for example, five minute from now. Absolute times indicate that execution should resume, for example, at three o'clock. Absolute times are a bit more cumbersome to use but allow periodic timing with no long term drift.

Asynchronous Timers

Asynchronous timing services cause notification at a given time. The notification can be delivered in any or all of three ways. First, one or more event flags within a single event flag group can be set. Second, a queue can be notified. Third, a specified subroutine can be run as a software interrupt.

Once set, an asynchronous timer remains in effect until it is either canceled or expires. A timer can be canceled, using the ID of the timer returned by the

microkernel when the timer was set, at any time prior to expiration. Expiration of the timer causes the notification to be delivered.

Asynchronous timers always specify absolute expiration times.

SUMMARY OF BENEFITS

Launching an application under the microkernel involves creating an address space, a kernel process and a task. The address space contains three areas: code, heap, and stack. The code area is write protected and mapped directly to the application file's code. No pre-loading of code is done at launch time; instead, the application faults itself in during execution. The heap and stack areas are backed by swapping space. They are spread apart within the logical address space to allow expansion as needed.

The application is free to allocate microkernel resources during its execution including other tasks. Application scheduling involves only the scheduling of tasks and is performed completely by the microkernel. Key OS and ToolBox routines can adjust the priority of the current task to ensure system responsiveness.

When applications run in separate address spaces, they cannot interfere with each other or with the microkernel. Gross application errors cannot corrupt the system or other applications. Upon termination, either normal or abnormal, the kernel process and all associated microkernel resources would be reclaimed.

Device management functions are largely subsumed by the message system. Device drivers are privileged software that service message ports. The Device Manager is used to resolve device names and return message object IDs. I/O requests are made by sending messages to the device objects. Synchronous and asynchronous I/O is provided by the message system without additional consideration on behalf of the device driver writer. Drivers no longer have any constraints regarding order of request processing or limitations regarding the number of concurrent requests processed at a time. Of course, writing drivers that handle multiple concurrent requests requires additional code.

Other than the microkernel, certain device drivers, and portions of the file system, no locked memory is required in the system. A much larger percentage of real memory is available to applications enabling better end-user perceived performance.

The Microkernel and Copland

The microkernel provides a software platform that is richer and more robust than System 7 and provides real value to software developers and end users. Full implementation of the microkernel will be achieved through a series of releases.

Copland is the first microkernel-based system. It has a memory model much like that presented by System 7.0 Virtual Memory. The use of separate address spaces is limited. All software that uses the existing toolbox is in a single 32-bit address space. Only a subset of the programming interface is used. All privileged code, including the microkernel, drivers, file system, and file system cache, is protected from direct access by any non-privileged software.

Subsequent releases will include larger subsets of the programming interface, including support for execution of applications in separate address spaces. These releases will require greater degrees of microkernel/OS/ToolBox integration.

USING THE MICROKERNEL API

All microkernel services are accessed through function calls. There are no exported data structures or low-memory locations. All microkernel interfaces are provided in C header files. The calling conventions required by the microkernel are those of the C runtime model used by Apple on the machine in question.

Most microkernel functions return an error indication that applications should check. The microkernel makes every effort to validate all parameters to each function call prior to doing any other work.

Many microkernel functions have “out” parameters. These are addresses that you pass to the function. The contents of the address are modified by the microkernel call. Passing null as the address of an out parameter tells the microkernel you don’t want that value returned. These null values do not generate an error. When a call to the microkernel fails, the microkernel attempts to clear any output parameters of that particular function. If the microkernel cannot return output parameters (for example, due to invalid pointers) but a call otherwise succeeds and has side effects, the microkernel returns a special error code. For more information, see the section “Errors,” later in this document.

Calling Conventions

Calls to the microkernel are performed using the shared library mechanisms of the Code Fragment Manager.

Note: Once control has transferred into the shared library, execution is considered to be within the microkernel. The shared library code, although within the address space of the client, is considered to be part of the microkernel’s implementation, which will not be documented and is subject to change at any time.

Stack Space

Most clients of the microkernel execute in user mode. These clients need not be concerned about the amount of stack space used by the microkernel because the microkernel’s execution never takes place on user mode stacks. Each user mode task has a separate microkernel stack that is used by the microkernel when the microkernel is called.

Privileged software, (such as drivers and the microkernel) always executes in supervisor mode. Clients of the microkernel must be aware that the microkernel does use stack space. The microkernel's design goals are to require less than 4K bytes of stack storage.

Certain circumstances can exhaust microkernel stack space. Acceptance functions run on the supervisor mode stack of their clients. It is the responsibility of these functions to perform stack checks before using stack storage. Stack checks can be performed by calling the RemainingStackSize service.

At certain times, the microkernel performs stack checks on behalf of its clients. If stack overflow is detected, a stack overflow exception is generated. Exceptions are described in the section "Exceptions" later in this document.

Addressing

Within the microkernel, all addressing is 32-bit clean. The upper or lower bits of an address must not be used for any purpose other than addressing.

MMU hardware and the contents of the underlying page tables are owned entirely by the microkernel. They must not be manipulated directly. The microkernel provides support for many of the operations that have historically required direct manipulation of the MMU. For descriptions of these services, see the section "Memory Management," later in this document.

SOME BASIC TYPES

This section introduces some basic types which are used throughout the API. They are presented in this section in no particular order.

Miscellaneous Types

The following type declarations are self-explanatory:

```
typedef UInt32   ByteCount;
typedef UInt32   ItemCount;
typedef SInt32   OSStatus;
typedef UInt32   OptionBits;
```

The symbol *kNilOptions* is provided for clarity.

```
enum
{
    kNilOptions    = 0
};
```

Parameter Block Versions

Any microkernel service that operates on a parameter block requires that you pass a parameter block version in the service's parameter list. In the future, this version number will allow the microkernel to provide backwards compatibility. Each parameter block type definition has an associated named version constant. As long as you always use the named constant your source is guaranteed to be correct and your object code will be supported.

```
typedef UInt32   PBVersion;
```

Duration

Many interfaces allow the caller to specify a time relative to the present. These values are of the type *Duration*.

```
typedef SInt32   Duration;
```

Values of type duration are 32-bits. They are interpreted in a manner consistent with the System 7 Time Manager as follows: positive values are in units of milliseconds, negative values are in units of microseconds. Therefore the value 1500 is 1,500 milliseconds or 1.5 seconds while the value -8000 is 8,000

microseconds or 8 milliseconds. Notice that many values can be expressed in two different ways. For example, 1000 and -1000000 both represent exactly one second. When two representations have equal value, they can be used interchangeably. Neither is preferred or inherently more accurate.

Values of type Duration can express times as short as one microsecond or as long as 24 days. However, two values of Duration are reserved and have special meaning. The value zero, (0) specifies exactly the present time. A value of 0x7FFFFFFF, the largest positive 32-bit value, specifies an infinite time from the present.

The following definitions are provided for use with values of type Duration:

```
enum
{
    durationMicrosecond = -1,
    durationMillisecond = 1,
    durationSecond      = 1000,
    durationMinute      = 1000 * 60,
    durationHour        = 1000 * 60 * 60,
    durationDay         = 1000 * 60 * 60 * 24,
    durationForever     = 0x7FFFFFFF,
    durationImmediate   = 0,
};
```

Time

The type *Nanoseconds* is another form for representing time. Values of this type represent an unsigned 64-bit integer.

```
typedef UnsignedWide    Nanoseconds;
```

A second data type is used to specify absolute times. These values are of the type *AbsoluteTime*. They are a microkernel-defined unit and are 64-bits in width. No assumptions can be made about what unit AbsoluteTime is based on.

```
typedef UnsignedWide    AbsoluteTime;
```

Note: Conversion routines are provided in the form of a shared library (Timing) to convert between Nanoseconds, Durations and AbsoluteTime.

IDs

IDs are used whenever you create, manipulate, or destroy a kernel object. There are specific ID types for each kernel object.

```
typedef struct OpaqueTaskID *           TaskID;
typedef struct OpaqueKernelProcessID *  KernelProcessID;
typedef struct OpaqueAddressSpaceID *   AddressSpaceID;
typedef struct OpaqueAreaID *           AreaID;
typedef struct OpaqueAreaReservationID * AreaReservationID;
typedef struct OpaqueIOPreparationID *  IOPreparationID;
typedef struct OpaqueSoftwareInterruptID * SoftwareInterruptID;
typedef struct OpaqueEventGroupID *     EventGroupID;
typedef struct OpaqueKernelQueueID *    KernelQueueID;
typedef struct OpaquePortID *           PortID;
typedef struct OpaqueObjectID *         ObjectID;
typedef struct OpaqueReceiveID *        ReceiveID;
typedef struct OpaqueMessageID *        MessageID;
typedef struct OpaqueTimerID *          TimerID;
typedef struct OpaqueBackingObjectID *  BackingObjectID;
```

The value *kInvalidID* is reserved to mean “no ID.”

```
enum
{
    kInvalidID    = 0
};
```

Iterators

The microkernel provides several iteration functions that allow the client to obtain the IDs of all kernel objects within a specified domain. For example, you can iterate over all the tasks within a given kernel process or all of the message objects associated with a given message port. An iteration call always returns a snapshot of the requested IDs.

Each of these functions takes a similar set of arguments and works in a similar way. For example, the function to return the IDs of all the tasks in a kernel process is:

```
OSStatus GetTasksInKernelProcess(KernelProcessID  theKernelProcess,
                                 itemCount         requestedTasks,
                                 itemCount *       totalTasks,
                                 TaskID *          theTasks);
```

Each iteration function provided by the microkernel requires at least three parameters. These are *requestedItems*, *totalItems*, and *theItems*.

- theItems points to storage where the microkernel will fill in the list of requested IDs. If requestedItems is zero, theTasks can be nil.
- requestedTasks specifies the number of entries for which space is available at the location pointed to by theItems.
- totalTasks points to a location in which the microkernel returns the number of items. If less than or equal to requestedTasks, all the items were returned. If greater than requestedTasks, there wasn't enough space in theTasks to return all the IDs. The function result is noErr in either case.

Generally, iteration functions should be called in a loop. If all IDs aren't returned, allocate a new buffer of size *totalTasks and try again. The number of tasks may (but probably won't) increase several times before all tasks are successfully retrieved.

For example:

```

TaskID *    theTasks      = nil;
ItemCount  totalTasks    = 0;
ItemCount  requestedTasks = 0;
Boolean    done          = false;
OSStatus   status        = noErr;

while (!done) {
    status = GetTasksInKernelProcess ( targetKernelProcess,
                                     requestedTasks,
                                     &totalTasks,
                                     theTasks);

    if (status == noErr) {
        if (totalTasks > requestedTasks) {
            if (theTasks != nil)
                PoolDeallocate (theTasks);
            theTasks = PoolAllocate (totalTasks);
            if (theTasks == nil) {
                status = memFullErr;
                done = true;
            }
            requestedTasks = totalTasks;
        }
        else
            done = true;
    }
    else
        done = true;
}

```

ERRORS

As with all system interfaces, you should check the OSStatus code returned by each microkernel service you call. OSStatus values are 32-bits wide. However, to remain compatible with System 7, all values currently returned by the microkernel are in the range of negative 16-bit values.

```
typedef SInt32 OSStatus;
```

Error codes returned by the microkernel fall into one of two categories. Some error codes are generic in nature and could be returned by nearly any microkernel service; these include paramErr or memErr. Other error codes are specific in nature and may only be returned by a specific service.

Whenever possible, the microkernel returns an error rather than causing an exception. However, in certain cases, erroneous calls to the microkernel may result in exceptions. For example, if you pass an invalid address you may receive either paramErr or incur an access violation exception.

Note: The microkernel makes no guarantee as to when exceptions may occur as opposed to returning paramErr, and the choice may change from one release to the next.

Generic Errors

Described here are the error codes that could be returned by any microkernel service. The meanings given apply only to the meaning of that error code when it is returned by a microkernel service. Other system software may use that same error code to indicate some other error.

- paramErr indicates that a parameter value is out of range or that a combination of parameters passed to the service are illegal.
- kernelReturnValueErr indicates that a requested operation was performed, but a return value could not be provided due to a problem accessing the caller's return value area.
- memFullErr indicates that the microkernel could not allocate the resources necessary to satisfy the service request.
- kernelPrivilegeErr indicates that the caller of a microkernel service is non-privileged and cannot use the service in question.

KERNEL PROCESS MANAGEMENT

Kernel processes are created through explicit requests to the microkernel. Kernel processes are deleted explicitly through a request or implicitly during task termination processing, when it is determined that the kernel process has no additional tasks.

Tasks that belong to a specific kernel process can allocate various microkernel resources including messages, event flag groups, message ports, message objects, other tasks, etc. If, at the time the kernel process is being deleted, these resources have not been deallocated, they will be reclaimed as part of the kernel process deletion process.

Kernel processes, when newly created, contain no tasks. Because automatic kernel process deletion is a side effect of task termination, kernel processes that never have tasks are never automatically deleted. If you create a kernel process you must be careful that you either create a task within that kernel process or explicitly delete the kernel process.

Deleting a kernel process causes the termination of all tasks within that kernel process and reclamation of all resources that belong to the kernel process. The kernel process may or may not be deleted by the time the `DeleteKernelProcess` function returns to the caller. A kernel process is never deleted until all its member tasks have terminated. Task termination is discussed in the section "Task Management," later in this document.

If, as the result of deleting a kernel process, that kernel process's address space is no longer accessible from any kernel process, the address space is also deleted. Address space deletion is discussed in the section "Address Space Management," later in this document.

Creating kernel processes

```
OSStatus CreateKernelProcess (KernelProcessName    theName ,
                              AddressSpaceID       theAddressSpace ,
                              KernelProcessID *    theKernelProcess);
```

`theName` specifies a four character name that may be useful to the kernel process's creator, e.g. for debugging purposes. This name is not used by the kernel and can be obtained by using `GetKernelProcessInformation`.

theAddressSpace is the ID of the address space that is to be addressable by the kernel process's tasks.

theKernelProcess is updated with the ID of the newly created kernel process.

Deleting kernel processes

```
OSStatus DeleteKernelProcess (KernelProcessID  theKernelProcess,  
                             TerminateOptions  options);
```

theKernelProcess is the ID of the kernel process to be deleted.

options is unused.

Obtaining The Current KernelProcessID

You can obtain the ID of the current kernel process whenever executing at task level.

```
KernelProcessID  CurrentKernelProcessID (void);
```

Obtaining Information About a Kernel Process

You can obtain information about a given kernel process by using the GetKernelProcessInformation service.

```
OSStatus GetKernelProcessInformation  
        (KernelProcessID  theKernelProcess,  
         PBVersion        version,  
         KernelProcessInformation * kernelProcessInfo);
```

```
typedef struct KernelProcessInformation  
{  
    KernelProcessName  name;  
    AddressSpaceID    addressSpace;  
};
```

The fields of the KernelProcessInformation record have the following meanings:

- name indicates the four-character name provided when the kernel process was created.

- `addressSpace` indicates the ID of the address space in which the kernel process resides. This address space was specified at the time the kernel process was created.

```
enum  
{  
    kKernelProcessInformationVersion = 1  
};
```

TASK MANAGEMENT

About Task Hierarchy

Tasks are found within conceptually enclosing environments called kernel processes. All tasks within a kernel process share the same address space. Additionally, tasks live within a parent-child hierarchy. Tasks with no parent are called *orphans*, and they live at a root of a task tree within their kernel process. By default, a task is the child of the task that caused its creation; a task's creator is called its *parent*. During task creation, you can specify that the created task be an orphan instead of a child.

The TaskRelationship type is used in conjunction with certain operations that affect more than one task.

```
typedef UInt32 TaskRelationship;
enum
{
    kTaskOnly          = 0,
    kTaskAndChildren  = 1,
    kTaskFamily       = 2,
    kTaskKernelProcess = 3
};
```

- kTaskOnly means just that.
- kTaskAndChildren means that the operation should be applied to the task and each of its children and each of their children, etc.
- kTaskFamily requires that the microkernel must first find the ancestor of the specified task which is an orphan and then perform the operation as if that orphan had been specified and the relationship had been kTaskAndChildren.
- kTaskKernelProcess causes the operation to be applied to each task within the kernel process of the specified task.

Tasking operations that can affect more than one task through use of a TaskRelationship do not operate on those tasks in any defined order.

About Task Scheduling

Tasks are scheduled for execution based only upon their CPU priority. No consideration is given to the kernel process to which a task belongs or to the

priorities of the task's parent or children. The initial priority of a task is specified by its creator but may be subsequently changed.

Tasks are scheduled with either a run-til-block or time-slice policy. Time slicing is only used to provide CPU time to tasks of equal priority. The execution of a higher priority task is never preempted to allow a lower priority task to execute. While the kernel doesn't currently perform any dynamic priority adjustments, it may do so in the future.

About Task Parameters and Results

When you create a task, you specify a subroutine that is to be executed within the context of the newly created task. That routine should conform to the TaskProc declaration.

```
typedef OSStatus (*TaskProc) (void * p);
```

The parameter *p* is specified at the time of creation and may be used for any purpose. The result returned by the task at the time it terminates will be returned to its creator as specified by the TerminationEvent parameter to the CreateTask call. If the task terminates abnormally, for example, due to an unhandled exception, the system provides a suitable result value.

About Task Termination

Task execution can be terminated either implicitly, when the main routine of the task returns, or explicitly through use of the TerminateTask service.

- Termination guarantees that no more task code will run once termination begins.
- Termination forcibly unblocks a blocked task. This may have side effects on system code that maintain data structures (such as open files) on behalf of the task being terminated.
- The termination process is irreversible. During the time required to terminate a task, the task is specially marked and all operations on it behave as if it were already terminated.

Once termination begins, a check is made to see if the terminating task has children. If children are present, no further actions are taken until they terminate. This allows children that reference their parent's resources (such as pointers into the stack) to continue normal execution. When no children remain, any

remaining resources including all stacks and control blocks internal to the microkernel are reclaimed.

Finally, the task's termination events, if any, are delivered. A termination event may be specified at task creation by the task's creator. Termination events are the only microkernel-provided method of learning of a task's termination. For a complete description of how termination event notifications are delivered, see the sections "Event Notification" and "Terminating A Specific Task," later in this chapter.

If, as the result of task termination, that task's kernel process contains no tasks, the kernel process is implicitly deleted. Kernel process deletion is discussed in the "Kernel Process Management" section of this document.

THE TASKING SERVICES

Tasks are always referenced by their IDs.

Creating Tasks

```
typedef OptionBits      TaskOptions;
typedef OSType          TaskName;
enum
{
    kTaskIsPrivileged      = 0x00800000,
    kTaskIsOrphan          = 0x00400000,
    kTaskPriorityMask       = 0x0000001F,
    kTaskPriorityIsAbsolute = 0x00000100,
    kTaskRaisePriorityBy    = 0x00000200,
    kTaskLowerPriorityBy    = 0x00000400,
    kTaskRaisePriorityToAtLeast= 0x00000800,
    kTaskLowerPriorityToAtMost = 0x00001000,
    kTaskPriorityIsSymbolic = 0x00002000,
    kTaskIsResident        = 0x00004000,
    kTaskAppPriority        = 0x00002000,
    kTaskAppNonUIPriority   = 0x00002001,
    kTaskHighServerPriority = 0x00002002,
    kTaskServerPriority     = 0x00002003,
    kTaskLowServerPriority  = 0x00002004,
    kTaskBackgroundPriority = 0x00002005,
    kTaskDriverPriority     = 0x00002006,
    kTaskHighRealTimePriority = 0x00002007,
    kTaskRealTimePriority   = 0x00002008,
    kTaskLowRealTimePriority = 0x00002009
};

OSStatus CreateTask (TaskName      name,
                    KernelProcessID owningKernelProcess,
                    TaskProc       entryPoint,
                    void *         parameter,
                    LogicalAddress  stackBase,
                    ByteCount       stackSize,
                    KernelNotification * terminationEvent,
                    TaskOptions     options,
                    TaskID *        theTask);
```

CreateTask creates a task subject to the parameters provided.

name specifies a four character name that may be useful for subsequent debugging. The name is stored by the microkernel in association with the task. This name is not used by the microkernel for any purpose and can be obtained using GetTaskInformation.

owningKernelProcess specifies an existing kernel process to which the task will belong. Tasks created in kernel processes other than that of the caller will be orphans within the specified kernel process.

entryPoint is the address of a subroutine and will become the initial PC of the task created. This address must be within the address space of the kernel process specified by the kernel process.

parameter is a single 32-bit parameter which will be passed to entryPoint when the task begins its execution. The value and interpretation of parameter are of no concern to the microkernel and may be used to convey information between the creator and created task.

stackBase is the optional address of memory to be used for the task's user mode stack. This parameter is ignored for privileged tasks. If the value is null, a stack will be created for the task by the microkernel. If non-null, the caller guarantees that stackSize bytes are available to the task at this address and will remain available for use by the task until it has terminated.

stackSize indicates the size of the stack desired for the task. Although the microkernel may detect certain stack overflow situations, it is the responsibility of the task to ensure it does not run out of stack space. Microkernel-detected stack overflows are converted into stack overflow exceptions. If the task is non-privileged, the stackSize parameter specifies the size of the user-mode stack. If the task is privileged, the stackSize parameter indicates the number of bytes of microkernel stack which should be allocated for the task; this value is in addition to the microkernel's requirements on this stack.

terminationEvent allows the creator to be notified upon the termination of the task being created. If a null value is passed, the creator is given no notification. If a terminationEvent is specified it is delivered at the time the task finishes its termination. If the notification specifies a status variable to set, the value provided is either that from the return statement of the main routine of the task or the value supplied in the TerminateTask call that caused the task to terminate. This value is also supplied as the second parameter to a software interrupt if the notification specifies a software interrupt. (For details, see the section "Software Interrupts," later in this document).

options is used to control various aspects of task creation. (TaskOptions is a mask that can be used to extract the priority field from the options word.)

- kTaskIsPrivileged, when set, causes the task to be privileged and to execute in supervisor mode.

- `kTaskIsOrphan`, when set, specifies that the task being created should not be a child of the creator, but rather should live at the root of the kernel process to which it belongs; this affects the termination relationship between the creator and createe.
- `kTaskIsResident` applies only if `stackBase` is null, and indicates that the microkernel-created task stack(s) must be resident.
- `kTaskPriorityIsAbsolute` indicates that the task's priority is specified by a number between 1 and 30 in the priority field of the options field, which is specified by the task priority mask. If this option is absent, a valid priority category must be specified. (Bits `kTaskLowerPriorityBy`, `kTaskRaisePriorityBy`, `kTaskRaisePriorityToAtLeast`, and `kTaskLowerPriorityToAtMost` should not be used when creating a task. They have meaning only when changing an existing task's priority. For details, see the section "Setting a Task's Execution Priority," later in this document.
- The priority field indicates the initial CPU priority of the task and is used by the scheduler. Either a priority category (the preferred method) or an absolute priority can be specified (if the `kTaskPriorityIsAbsolute` option is set). Specifying a priority of zero causes the created task to inherit the priority of its creator.

Absolute CPU priorities range from 1 to 30 with larger numbers signifying higher scheduling priority. The meaning and behavior of a specific priority number may change from one system release to another, and certain absolute priorities may be adjusted dynamically by the microkernel. Absolute priorities should be avoided, because there's no way to determine how an absolute priority relates to the priority categories (for example, 15 may be higher or lower than `kTaskAppPriority`), and this relationship may change from one system release to the next.

Priority categories specify the priority of the task relative to other tasks in terms of the task's intended use. The absolute priority a task of a given category receives may change from one system release to another, and tasks of certain priority categories may have their absolute priorities adjusted dynamically by the microkernel. Priority categories are:

- `kTaskAppPriority` is the priority with which all application tasks are initially created. Most application tasks that aren't intended for background computation should be created with this priority.
- `kTaskAppNonUIPriority` is slightly lower than `kTaskAppPriority`. This priority is suitable for such things as document repagination or spreadsheet recalculation that should proceed whenever user interactions don't need the CPU, but should preempt background computations of `kTaskBackgroundPriority`.
- `kTaskServerPriority` is higher than application priorities and is the priority at which most system services should execute.
- `kTaskHighServerPriority` and `kTaskLowServerPriority` are also higher than application priorities but slightly higher and lower, respectively, than `kTaskServerPriority`.

- `kTaskBackgroundPriority` is a priority that allows tasks to soak up otherwise unused CPU time. Most of the total CPU time will usually be available for tasks of this priority, but if an application or system service needs the CPU, it will preempt `kTaskBackgroundPriority` tasks. Compressing files and performing full-text indexing are good candidates for this priority level.
- `kTaskDriverPriority` is meant for device drivers. It is higher than anything except `kTaskRealTimePriority`. Drivers should get control whenever an I/O request is made so that they can start the I/O as quickly as possible. A driver will then typically block while the I/O operation proceeds, thus allowing maximum overlap of computation and I/O. If any task that does I/O is of higher priority than the driver serving that I/O, overlap between its computation and I/O will be greatly reduced.
- `kTaskRealTimePriority` is for tasks which must preempt most other tasks in the system, including all other applications, device drivers and system services. `kTaskHighRealTimePriority` is slightly higher than `kTaskRealTimePriority`, and `kTaskLowRealTimePriority` is slightly lower. `kTaskHighRealTimePriority` is the highest possible priority in the system, and preempts all other tasks. Tasks at real time priority levels degrade the throughput of system services and I/O.

Extended computations should be done only at priorities of `kTaskAppPriority`, `kTaskAppNonUIPriority`, or `kTaskBackgroundPriority`. If a server needs to perform an extended computation, it should create another task of lower priority, or use the `SetTaskPriority` service to lower its own priority.

`theTask` is set to the ID of the newly created task.

Terminating a Specific Task

```
OSStatus TerminateTask (TaskID          theTask,
                        TaskRelationship scope,
                        TerminateOptions options,
                        OSStatus        status);
```

`TerminateTask` forces one or more tasks to terminate.

`theTask` is the task ID of the task to be terminated.

`scope` indicates what other tasks should also be terminated.

`options` is presently unused.

`status` may be returned in any termination events for tasks specified by `TerminateTask` (but only for those tasks that have not yet begun to terminate at the time of the call).

For additional information about terminating a task, see the section “About Task Termination,” earlier in this document.

Obtaining the ID of the Current Task

```
TaskID CurrentTaskID (void);
```

CurrentTaskID returns the ID of the current task.

Determining the Amount of Available Stack Space

```
ByteCount RemainingStackSize (void);
```

RemainingStackSize returns the amount of stack space available on the current stack. It may be called from any execution level.

Obtaining Information About a Task

You can obtain information about a given task by using the GetTaskInformation service. The information returned reflects the state of the task at the time the GetTaskInformation service is made. Due to the preemptive nature of the microkernel, this information may be obsolete even before the GetTaskInformation service returns to its caller. The information that is available is returned in the form of a TaskInformation record with the following type definition:

```
typedef OSType SchedulerState;

typedef struct TaskInformation
{
    TaskName          name;
    KernelProcessID  owningKernelProcess;
    TaskPriority      options;
    SchedulerState    taskState;
    SchedulerState    swiState;
    Boolean           isTerminating;
    Boolean           reserved2[3];
    ItemCount        softwareInterrupts;
    LogicalAddress    stackLimit;
    ByteCount        stackSize;
    AbsoluteTime     creationTime;
    AbsoluteTime     cpuTime;
    void *           reserved;
```

```
} TaskInformation;
```

The various fields of the TaskInformation record have the following meanings:

- name indicates the four-character name provided when the task was created
 - owningKernelProcess indicates the ID of the kernel process to which the task belongs
 - options indicates the current absolute CPU priority of the task together with the originally specified task options
- (**Note:** in general the value of the priority field may NOT be the same as the one specified at the time the task was created or when SetTaskPriority was last called for it. This is because priorities may be specified symbolically as input arguments to microkernel routines, but they are always returned as absolute values.)
- taskState is a four-character abbreviation of the scheduler state of the task
 - swiState is a four-character abbreviation of the scheduler state of software interrupts for the task
 - isTerminating, if true, indicates that the task is in the process of terminating
 - softwareInterrupts indicates the number of software interrupts that have been processed by the task. It does not indicate how many software interrupts are pending execution by that task.
 - stackLimit is the logical address of the end (lowest point) of the task's stack.
 - stackSize is the total number of bytes in the task's stack.
 - creationTime indicates the time at which the task was created. Subtracting this value from the value returned by the UpTime service will yield the amount of wall-clock time that has passed since the task was created.
 - cpuTime indicates the amount of CPU time that the task has consumed. This includes all task execution time in the microkernel as well as that consumed by processing software interrupts. Also included is the time spent processing hardware and secondary interrupts incurred while the task was running.

```
enum  
{  
    kTaskInformationVersion = 1  
};
```

```
OSStatus GetTaskInformation (TaskID          theTask,  
                             PBVersion      version,  
                             TaskInformation * taskInfo);
```

GetTaskInformation returns information about the specified task to the caller.

theTask specifies the ID of the task about which information is to be returned.

version specifies the version number of TaskInformation to be returned. This provides backwards compatibility. kTaskInformationVersion is the version of TaskInformation defined in the current interface.

taskInfo is the address of a TaskInformation record. This record is filled in by the microkernel with information about the designated task.

Setting a Task's Execution Priority

You can alter the priority of a task. Note that the priority of a task does not change until it is next made eligible to execute. This means, for example, that a lower priority task that is waiting for an event flag will not have its priority in the wait queue adjusted until after it has acquired the flag or until the wait operation has timed out. The effect of SetTaskPriority is seen immediately by GetTaskInformation even though the task's priority change may not as yet have taken effect. The priority categories are described in "Creating a Task," earlier in this document.

```
enum
{
    kTaskPriorityMask           = 0x0000001F,
    kTaskPriorityIsAbsolute     = 0x00000100,
    kTaskRaisePriorityBy        = 0x00000200,
    kTaskLowerPriorityBy        = 0x00000400,
    kTaskRaisePriorityToAtLeast = 0x00000800,
    kTaskLowerPriorityToAtMost  = 0x00001000,
    kTaskAppPriority            = 0x00002000,
    kTaskAppNonUIPriority       = 0x00002001,
    kTaskHighServerPriority     = 0x00002002,
    kTaskServerPriority         = 0x00002003,
    kTaskLowServerPriority      = 0x00002004,
    kTaskBackgroundPriority     = 0x00002005,
    kTaskDriverPriority         = 0x00002006,
    kTaskHighReadTimePriority   = 0x00002007,
    kTaskRealTimePriority       = 0x00002008,
    kTaskLowRealTimePriority    = 0x00002009
};

OSStatus SetTaskPriority (TaskID          theTask,
                        TaskRelationship scope,
                        TaskOptions      options);
```

theTask specifies the task whose priority is to be changed.

scope specifies what other tasks should be affected.

options specifies the new task priority and options. Its format is similar to the options parameter to CreateTask, except that only priority-related options can be specified.

- kTaskPriorityIsAbsolute indicates that the task's priority is specified by a number between 1 and 30 in the priority field of the options. If this option is absent, the priority field must contain a valid priority category.
- kTaskRaisePriorityBy indicates that the task's priority is to be boosted by the amount specified in the priority field. The new priority is limited to the range 1-30. The kTaskPriorityIsAbsolute option must also be specified.
- kTaskLowerPriorityBy indicates that the task's priority is to be reduced by the amount specified in the priority field. The new priority is limited to the range 1-30. The kTaskPriorityIsAbsolute option must also be specified.
- kTaskRaisePriorityToAtLeast is used to set the new priority to the greater of the current priority and the specified priority. That is, the new priority will be at least the value specified. The specified priority may be either absolute (if the kTaskPriorityIsAbsolute option is set) or symbolic.
- kTaskLowerPriorityToAtMost is used to set the new priority to the lower of the current priority and the specified priority. That is, the new priority will be at most the value specified. The specified priority may be either absolute (if the kTaskPriorityIsAbsolute option is set) or symbolic.
- The priority field contains either an absolute priority (if the kTaskPriorityIsAbsolute option is set), a symbolic value (if the kTaskRaisePriorityToAtLeast or kTaskLowerPriorityToAtMost options are used and the kTaskPriorityIsAbsolute option is not set), or a delta value (if the kTaskRaisePriorityBy or kTaskLowerPriorityBy options are set).

Note: The toolbox may boost priorities of all tasks in a kernel process when that application is brought to the front, and reduce the priority again when the application is moved away from the front. Any changes an application makes to its tasks' priorities (by calling SetTaskPriority) may be modified at these times. Applications should rely only on the relative task priorities within their kernel process, not on their task priorities relative to other tasks in the system. Privileged mode task priorities aren't modified by the system.

Iterating Over Task IDs

```
OSStatus GetTasksInKernelProcess(KernelProcessID  theKernelProcess,
                                itemCount         requestedTasks,
                                itemCount *       totalTasks,
                                TaskID *         theTasks);
```

GetTasksInKernelProcess allows the caller to find the IDs of all tasks within a particular kernel process. For additional information about using iteration functions see the section, "Some Basic Types," earlier in this document.

theKernelProcess indicates the kernel process of interest.

requestedTasks indicates the maximum number of task IDs that should be returned. This indicates the number of entries available at the location pointed to by theTasks.

totalTasks is filled in with the total number of tasks within the kernel process. If less than or equal to requestedTasks, all tasks were returned; if greater than requestedTasks, insufficient space was available to return all taskIDs.

theTasks is filled in with the IDs of the tasks within the kernel process.

The status returned is:

- noErr if the kernel process exists, whether or not all task IDs were returned.
- kernelIDerr if the kernel process doesn't exist

EXCEPTIONS

Exceptions are synchronous alterations in program flow that arise from exceptional conditions caused by normal instruction execution. Certain exceptions, such as page faults, are handled entirely by the microkernel. Other exceptions, such as illegal instruction, are presented to the client for resolution; the handling of these exceptions by the client is covered in this section.

Different processor families support different exception models and cause exceptions under varying circumstances. Exceptions are, therefore, processor specific in nature. The kinds of exceptions, the information made available at the time of the exception, and the ability to resume execution after an exception are all processor specific. The microkernel isolates these processor dependencies by presenting a processor independent model for the registration and invocation of exception handlers. However, exception handlers that want to correct an exception (instead of simply reporting it) must be processor dependent.

Exceptions can arise during processing at any execution level: task, secondary interrupt, or hardware interrupt. This section covers only exceptions that occur during task level execution. For details about handling exceptions during non-task level execution, see the sections "Interrupt Handling" and "Secondary Interrupt Handlers," later in this document.

About Exception Handlers

The microkernel provides support for catching, resolving, and proceeding from exceptions, subject to the specifics of the processor. Exception handling is performed within the context of the task that incurred the exception. Exception handlers are installed for a given task and do not inherently affect other tasks in any way.

Exception handlers are not nested. Each task can have only a single handler. Installing an exception handler overrides any previous exception handler installed for that task. When a handler is installed, the previous handler for that task is returned. This allows a routine to temporarily install an exception handler and then restore the previous handler.

At the time of an exception, the exception handler is provided with information about the nature of the exception and the state of the processor at the time of the exception. The type `ExceptionInformation` is machine dependent and will be described in the microkernel implementation guide for each product.

Exception handlers may resume execution either by returning to the microkernel or by transferring control using `longjmp` or similar mechanisms. If a handler chooses to return to the microkernel, it must supply a result indicating the action the microkernel should take. A value of `noErr` indicates that the exception has been resolved and that execution should resume based on information provided by the handler. Any other value indicates that the exception handler could not resolve the exception and that the task should be terminated.

Examples of actions an exception handler might take include:

- Performing a `longjmp` back to the application's main event loop.
- Changing some register values in the `ExceptionInformation` and returning with a status of `noErr`, thus retrying the operation that caused the exception with new values in the registers.
- Changing some register values in the `ExceptionInformation`, incrementing the PC in the `ExceptionInformation`, and returning a status of `noErr`, thus substituting the new register values for those that would have been computed by the operation that caused the exception.
- Returning an error status, indicating the task should be terminated.

Implementation Note:

The reason these possibilities all work is that there's no exception-related state kept in the kernel when an exception handler is being run — all the state associated with the exception is in the `ExceptionInformation` record, and that's on the stack with the exception handler. If the exception handler pops the stack back to some previous state and jumps to another location, the kernel has no memory that an exception occurred. If the handler returns, it actually returns to glue code that re-enters the kernel. The kernel then copies the exception information from the stack, loads those values into the CPU registers, and resumes execution of the original environment with those register values.

Exceptions Within Exception Handlers

Exception handlers are invoked on the stack of the task that caused the exception and synchronously to that task's execution. Exception handlers that incur exceptions cause exception processing to begin recursively. Exception handlers may be preempted by a software interrupt, which may in turn cause an exception that leads to the invocation of an exception handler yet again.

Exception Handler Declarations

All exception handlers should conform to the ExceptionHandler declaration.

```
typedef OSStatus (*ExceptionHandler)  
                (ExceptionInformation * theException);
```

Installing Exception Handlers

The microkernel does not provide any implicit exception handling. Tasks that incur exceptions and have not installed an exception handler are terminated when an exception occurs.

```
ExceptionHandler InstallExceptionHandler (ExceptionHandler theHandler);
```

theHandler specifies a subroutine which becomes the active exception handler for the task. Specifying null indicates that no exception handler should be installed. The previously active handler is returned as the function result.

Software interrupt handlers may call InstallExceptionHandler. An exception handler installed this way will remain in effect only for the duration of the software interrupt handler which installed it.

Hardware interrupt handlers, secondary interrupt handlers, and acceptance functions should not call InstallExceptionHandler. Exception handlers for such functions are specified at the time the function is installed.

SOFTWARE INTERRUPTS

The software interrupt mechanism allows a given subroutine to be run asynchronously to a given task's normal flow of control yet still be within the context of that task. Software interrupts are said to be *sent* to a task by either a different task, a secondary interrupt handler, or in some cases, the task itself. Once sent, software interrupts are said to be *pending* until actually *activated*.

The execution of a software interrupt happens on the same stack and with the same addressing context from which the task typically executes. Software interrupts can be handled by a task even when its normal execution has been suspended. At completion of a software interrupt, the interrupted task will resume execution at the point of interruption. If the task was not executable prior to delivery of the software interrupt it will, again, become blocked upon whatever event it was awaiting prior to the interruption.

Software interrupts are serialized. If a task is executing a software interrupt routine and is sent a second software interrupt, it will finish processing the first interrupt prior to processing the second interrupt. This is true even if the first software interrupt handler performs some blocking operation such as waiting for an event flag or initiating a synchronous I/O operation.

The presence of a pending software interrupt or the invocation of a software interrupt handler does not inherently change the execution priority of the associated task or affect the scheduling of that task or any other tasks in any way.

Controlling Software Interrupts

You can call `EnableSoftwareInterrupts` and `DisableSoftwareInterrupts` to enable and disable software interrupts. These operations nest automatically, so every call to `DisableSoftwareInterrupts` must be matched by a call to `EnableSoftwareInterrupts`. Calls to either of these services have no effect when the task is processing a software interrupt.

```
void DisableSoftwareInterrupts (void);  
void EnableSoftwareInterrupts (void);
```

Querying the Level of Execution

A task can determine whether it is executing at software interrupt level by calling `InSoftwareInterruptHandler`.

```
Boolean InSoftwareInterruptHandler (void);
```

A software interrupt handling routine must conform to the prototype definition `SoftwareInterruptHandler`. The parameters, `p1` and `p2`, specify the creator and sender of the particular software interrupt, respectively.

```
typedef void (*SoftwareInterruptHandler) (void *   p1,  
                                         void *   p2);
```

Specifying Software Interrupts

Software interrupts are specified by a software interrupt ID. These IDs are created by calling `CreateSoftwareInterrupt`. Software interrupt IDs are instances of potential interrupt requests. The ID of a software interrupt is valid until it is released by the invocation of the software interrupt handler (which occurs sometime after a call to `SendSoftwareInterrupt`) or by deleting it by calling `DeleteSoftwareInterrupt`.

```
OSStatus CreateSoftwareInterrupt  
          (SoftwareInterruptHandler handler,  
           TaskID                  task,  
           void *                   p1,  
           Boolean                  persistent,  
           SoftwareInterruptID *    theSoftwareInterrupt);
```

`handler` is the routine address of the software interrupt handling routine. The address of this routine must be within the kernel process of the caller.

`task` is the ID of the task that will receive the software interrupt. If `task` is null, the current task will receive the interrupt. This task must be within the same kernel process as the calling task.

`p1` is the value that will be passed to `handler` as its `p1` parameter.

`persistent` indicates whether the ID of the software interrupt should be consumed when the software interrupt is activated or should persist until explicitly deleted by `DeleteSoftwareInterrupt`. A persistent software interrupt can be sent multiple times but only once per activation; that is, the software interrupt must run before it can be re-sent. For details, see the next section "Sending Software Interrupts."

`theSoftwareInterrupt` is updated with the ID of the created software interrupt.

Sending Software Interrupts

You can send a software interrupt to a specific task by calling `SendSoftwareInterrupt`. The software interrupt will be activated when the designated task becomes eligible for execution with software interrupts enabled and all software interrupts previously sent to the designated task have been processed. Sending a single software interrupt more than once results in an error. Persistent software interrupts can only be re-sent after they have been activated.

```
OSStatus SendSoftwareInterrupt
    (SoftwareInterruptID theSoftwareInterrupt,
     void *               p2);
```

`theSoftwareInterrupt` specifies a software interrupt previously created by `CreateSoftwareInterrupt`.

`p2` is the value that will be passed to the handler as its `p2` parameter.

Deleting a Software Interrupt

You can delete a software interrupt by calling `DeleteSoftwareInterrupt`. The software interrupt and its ID will be consumed immediately. Software interrupts that are pending can be deleted; they will never be activated.

```
OSStatus DeleteSoftwareInterrupt
    (SoftwareInterruptID theSoftwareInterrupt);
```

HARDWARE INTERRUPTS

About Interrupt Handlers

The microkernel provides support for installing and removing hardware interrupt handlers. Interrupt handlers are invoked by the microkernel in response to an external interrupt. Interrupt handlers execute on a special stack dedicated to interrupt processing. Interrupt handlers must operate within the restrictions of the interrupt execution model by not causing page faults, and not using certain system services.

To ensure maximum system performance, interrupt handlers should perform only those actions that must be synchronized with the external device that caused the interrupt and then queue a secondary interrupt handler to perform the remainder of the work associated with the interruption.

Microkernel services pertaining to hardware interrupts are only available to privileged clients.

Designating Interrupt Sources

Interrupts are identified by a 32-bit vector number. This number is not related in any way to the interrupt vector defined by the processor. These vector numbers will be assigned by Apple and described in the microkernel implementation guide for each product.

```
typedef UInt32    InterruptVector;
```

Exceptions Caused by Interrupt Handlers

Whenever you register an interrupt handler, you can specify an exception handler. That exception handler will gain control should an exception be incurred by the interrupt handler. Additional exception handlers cannot be installed nor can any exception handlers be removed during processing at hardware interrupt level.

When an exception occurs, control will be transferred to the exception handler as described in the section "Exceptions," earlier in this document. The handler can transfer control using `longjmp` or similar mechanisms, or it can return to the microkernel. If the handler returns to the microkernel indicating that the exception was handled, control will resume according to the exception state

information provided by the handler. If, however, the handler returns to the microkernel with status indicating that the exception was not handled, the system will crash.

When you register an interrupt handler, you can request that no exception handler be installed during the activation of that interrupt handler. Specifying a null value causes no exception handler to be installed. If an exception occurs with no exception handler installed, the system will crash.

Execution Context of Interrupt Handlers

Interrupt handlers are invoked with the appropriate addressing context (TOC) as determined at the time they are installed. However, all data and code references generated during the processing of a hardware interrupt must be to pages that are physically resident. Access to non-resident pages causes access error exceptions.

Arbitrating for Interrupts

It is not possible to usurp control of an interrupt that is already under control of some other handler. The policies for such arbitration are beyond the scope of the microkernel.

Parameters to Interrupt Handlers

When an interrupt handler is invoked, it is supplied with two 32-bit parameters. The first parameter indicates the source of the interruption and is the same vector number supplied at installation time. This allows a single interrupt handler installed for multiple sources to determine the source of the current invocation. The second parameter is a reference constant value that is used internally.

SECONDARY INTERRUPT HANDLERS

Secondary interrupt handlers are the primary synchronization mechanism used within the microkernel and its extensions. Secondary interrupt handlers must conform to the interrupt execution environment rules. These rules include no page faults and severe restrictions on the use of system services. Secondary interrupt handlers run on a special stack reserved just for this purpose. They cannot make any presumptions about the task context in which they execute.

The special characteristic of secondary interrupt handlers that makes them useful is the microkernel's guarantee that at most one handler is active at any time. This means that if you have a data structure that requires complex update operations and each of the operations utilizes secondary interrupt handlers to access or update the data structure, all access to the data structure will be atomic even though hardware interrupts are enabled during the access.

Although interrupts are taken during the execution of secondary interrupt handlers, no task level execution takes place on a uniprocessor system, which can lead to severely degraded system responsiveness. For that reason, the secondary interrupt handlers should be used only when absolutely necessary. It is always preferable to execute entirely at task level and use locking mechanisms as described in the synchronization ERS.

Note: When a secondary interrupt handler executes on a multiprocessor system, task level execution will continue on other CPUs. While the microkernel guarantees that all secondary interrupt handlers are serialized, it does not guarantee that no tasks run while a secondary interrupt handler runs. Any code that depends upon this behavior will work correctly on a uniprocessor system but fail on a multiprocessor system.

Microkernel services pertaining to secondary interrupts are available only to privileged clients.

About Secondary Interrupt Handlers

Secondary interrupt handlers are simple procedures. They have two parameters and return a status result.

```
typedef OSStatus (*SecondaryInterruptHandler2) (void * p1,  
                                               void * p2);
```

Exceptions in Secondary Interrupt Handlers

Whenever you queue or call a secondary interrupt handler, you can specify an exception handler. That exception handler will gain control should an exception be incurred by the secondary interrupt handler. Additional exception handlers cannot be installed nor can any exception handlers be removed during processing at secondary interrupt level.

Should an exception arise, control will be transferred to the exception handler as described in the section "Exceptions," earlier in this document. The handler can transfer control using `longjmp` or similar mechanisms, or it can return to the microkernel. If the handler returns to the microkernel indicating that the exception was handled, control will resume according to the exception state information provided by the handler. If, however, the handler returns to the microkernel with a status indicating that the exception was not handled, the system will crash.

When queuing or calling a secondary interrupt handler, you may request that no exception handler be installed during the activation of that secondary interrupt handler. Specifying a `nil` value causes no exception handler to be installed. If an exception occurs with no exception handler installed, the system will crash.

Queuing Secondary Interrupt Handlers

Queuing secondary interrupt handlers is usually done during the processing of a hardware interrupt. The secondary interrupt handler's execution will be deferred until execution is about to transition back to task level. You may, however, queue secondary interrupt handlers from secondary interrupt level. In this case, the enqueued handler will be run after all other such queued handlers, including the current handler, have finished executing. Only one flavor of secondary interrupt handler, those with two parameters, may be queued. You must specify the values of the two parameters at the time you queue the handler.

```
OSStatus QueueSecondaryInterruptHandler
    (SecondaryInterruptHandler2    theHandler,
     ExceptionHandler              theExceptionHandler,
     void *                        p1,
     void *                        p2);
```

Secondary interrupt handlers that are queued from hardware interrupt handlers consume microkernel resources from the time they are queued until the time they begin to execute. Since the microkernel can't allocate these resources dynamically at interrupt level, you must call `AdjustSecondaryInterruptHandlerLimit` at task level to allocate them. You

should do this once during your software's initialization, not each time you need to queue a secondary interrupt handler.

There are three operations that, when executed from hardware interrupt level, cause a secondary interrupt handler to be queued: QueueSecondaryInterruptHandler, SetEvents, and ClearEvents.

For each simultaneously queued secondary interrupt handler you use, you should increment the queued SIH limit by one. For example, if you queue a secondary interrupt handler from your hardware interrupt handler and don't queue any additional secondary interrupt handlers until the first has run, you only need to increment the queued SIH limit by one. When your software terminates or is removed, it should decrement the queued SIH limit.

Note: If you fail to increment the queued SIH limit, your software may appear to work fine. However, if an attempt is made to queue more secondary interrupt handlers than allowed by the limit, a QueueSecondaryInterruptHandler call may fail with memFullErr. Because the limit manipulated by AdjustSecondaryInterruptHandlerLimit is shared by all callers of QueueSecondaryInterruptHandler, the software that receives a bad status from QueueSecondaryInterruptHandler will not necessarily be the software that failed to call AdjustSecondaryInterruptHandlerLimit.

When you increment or decrement the queued SIH limit, the new value of the limit is returned.

```
OSStatus AdjustSecondaryInterruptHandlerLimit
        (long amount,
         unsigned long * newLimit);
```

Calling Secondary Interrupt Handlers

Secondary interrupt handlers can be called synchronously through use of the CallSecondaryInterruptHandler2 routine. This service may be used from either task level or secondary interrupt level but not from hardware interrupt level. The secondary interrupt handler is invoked immediately in response to calls to this service; it is never queued.

```
OSStatus CallSecondaryInterruptHandler2
        (SecondaryInterruptHandler2 theHandler,
         ExceptionHandler theExceptionHandler,
         void * p1,
```

```
void *                p2);
```

EVENT FLAGS

Event flags are primarily used for synchronizing operations among tasks and are similar to binary semaphores. Event flags come in groups, each group containing 32 separate flags or semaphores. The microkernel provides a set of operations that creates and deletes event groups and operates upon one or more of the flags within a specified group. As with most abstract types provided by the microkernel you cannot get access to the underlying data of an event group; it is maintained within the microkernel's address space and you may only reference it by the ID returned when the group is created.

In addition to operations that allow the creation and deletion of event flag groups, other operations allow you to set, clear, test, and wait for one or more flags within a group. These operations require you to specify a mask value that is used to manipulate the flags within the group.

```
typedef UInt32      EventMask;
```

Creating Event Flag Groups

```
OSStatus CreateEventGroup (EventGroupID * theGroup);
```

CreateEventGroup creates an event flag group and returns the group's ID. Each flag within the group is cleared. The event flag group will persist until it is explicitly deleted.

Deleting Event Flag Groups

```
OSStatus DeleteEventGroup (EventGroupID theGroup);
```

DeleteEventGroup destroys the specified event flag group. Any tasks waiting on flags within the group are made executable and the result of their wait operation will be a kernellncompleteErr.

Setting Event Flags

```
OSStatus SetEvents (EventGroupID  theGroup,  
                  EventMask      mask);
```

SetEvents sets specified flags in the specified event flag group. As a result of setting flags, tasks waiting on those flags will become eligible for execution. For

details, see the section “Waiting for Event Flags to Become Set,” later in this document.

theGroup specifies the event flag group.

mask specifies zero or more event flags to set.

Clearing Event Flags

```
OSStatus ClearEvents (EventGroupID  theGroup,  
                     EventMask     mask);
```

ClearEvents clears specified flags in the specified event flag group. Clearing event flags does not have any scheduling side effects.

theGroup specifies the event flag group.

mask specifies zero or more event flags to clear.

Examining the Value of Event Flags

```
OSStatus ReadEvents (EventGroupID  theGroup,  
                   EventMask *    currentValue);
```

ReadEvents returns the values of the event flags in the specified event flag group.

theGroup specifies the event flag group.

currentValue specifies where to return the values of the event flags.

Waiting for Event Flags to Become Set

```
typedef UInt32  EventFlagOperation;  
enum  
{  
    kEventFlagAll           = 0,  
    kEventFlagAny          = 1,  
    kEventFlagAllClear     = 2,  
    kEventFlagAnyClear     = 3,  
    kEventFlagSharedClear = 4  
};
```

```

OSStatus WaitForEvents (EventGroupID      theGroup,
                        Duration           timeout,
                        EventMask          mask,
                        EventFlagOperation operation,
                        EventMask *        value);

```

WaitForEvents waits for the flags specified by mask to become set within theGroup. If you want to wait for any one of several flags to become set use the kEventFlagAny operation. If you want to wait for multiple flags to all become set use the kEventFlagAll operation.

You can optionally cause the flags for which you were waiting to be cleared by using either kEventFlagSharedClear, kEventFlagAllClear or kEventFlagAnyClear rather than kEventFlagAny or kEventFlagAll respectively. For a detailed description of the effect of these options, see the next section, "The Processing of SetEvents."

The maximum amount of time spent waiting is controlled by the timeout parameter and may range from zero to infinite. The value field represents the value of the flags when either the condition is satisfied or timeout is exceeded.

The Processing of SetEvents

SetEvents can cause one or more task-scheduling operations to occur if any corresponding WaitForEvents requests become satisfied.

At the time of a set operation, zero or more tasks are waiting for flags with the specified group to become set. After the flag group has been updated to reflect the effect of the set operation, the waiting tasks are scanned in priority order. Higher priority tasks become eligible before lower priority tasks. Within a single priority, tasks that have been waiting longer are considered prior to those which have been waiting for a shorter time. If the condition specified by a given wait request is satisfied, the task is made executable and it is removed from the list of waiting tasks. Otherwise, the task remains on the list.

If the condition is satisfied and kEventFlagAnyClear or kEventFlagAllClear was specified, the effect of the clear operation happens before any other tasks' conditions are evaluated. If all waiting tasks specify such an option, exactly one eligible task is unblocked. If no tasks specify one of these options, all eligible waiting tasks are unblocked. kEventFlagSharedClear behaves like kEventFlagAnyClear, but the clear operation is deferred until all tasks have been unblocked. If all waiting tasks specify this option, the operation behaves like a "barrier": all tasks waiting at the time of the SetEvents are unblocked, but tasks

that subsequently call WaitForEvents with the same options will block (until another SetEvents operation occurs).

KERNEL QUEUES

Kernel queues, like event groups, are primarily used for synchronizing operations among tasks. The microkernel provides a set of operations that create and delete kernel queues and operate upon a specified kernel queue. As with most abstract types provided by the microkernel you cannot get access to the underlying data of a kernel queue; the queue is maintained within the microkernel's address space and you can only reference it by the ID returned when the kernel queue is created. A given kernel queue is associated with the team that created it. When the creating team is deleted, all associated kernel queues are deleted, which unblocks any waiting tasks.

In addition to operations that allow the creation and deletion of kernel queues, other operations allow you to notify and wait for a kernel queue to be notified. These operations require that you specify a kernel queue ID, and optional data associated with a kernel queue.

When a kernel queue is notified, it is given three 32-bit values. These values are delivered to the task that waits on the queue.

Creating Kernel Queues

```
OSStatus CreateKernelQueue (KernelQueueOptions  options,  
                             KernelQueueID *    theQueue);
```

CreateKernelQueue creates a queue and fills in theQueue. The queue will persist until it is explicitly deleted.

options specifies how the queue is created. Currently, no options are defined.

theQueue returns the kernel queue ID for the newly created queue.

CreateKernelQueue returns:

- noErr if the queue was successfully created.
- Other microkernel errors may be returned.

Deleting Kernel Queues

```
OSStatus DeleteKernelQueue (KernelQueueID theQueue);
```

DeleteKernelQueue destroys the specified kernel queue. Any tasks waiting on the kernel queue are made executable and the result of their wait operation will be a kernelIncompleteErr.

theQueue specifies the targeted queue.

DeleteKernelQueue returns:

noErr if the queue was successfully deleted.
Other microkernel errors may be returned.

Notifying a Kernel Queue

```
OSStatus NotifyKernelQueue (KernelQueueID      theQueue,  
                             void *            p1,  
                             void *            p2,  
                             void *            p3);
```

When a kernel queue is notified, the values of p1, p2, and p3 are copied.

If there is a task waiting on the kernel queue at the time of notification, the task will be delivered the queue data and made executable. If there is more than one waiting task, the data is delivered to the task that has waited the longest. (Unlike event flags, kernel queue notification is first-in / first-out without regard to task priority.) If no tasks are waiting on the kernel queue, the data will be enqueued for the next task that waits on the kernel queue.

theQueue specifies the targeted queue.

p1 is user-specified data.

p2 is user-specified data.

p3 is user-specified data.

NotifyKernelQueue returns:

noErr if the data was successfully queued or delivered.
Other microkernel errors may be returned.

Notifying a Kernel Queue from Secondary Interrupt Level

Each entry put into a kernel queue consumes microkernel resources from the time it is placed in the queue with NotifyKernelQueue until the time it is retrieved with WaitOnKernelQueue. Since the microkernel can't allocate these

resources dynamically at secondary interrupt level, you must call `AdjustKernelQueueSIHLimit` at task level to allocate them. You should do this once during your software's initialization, not each time you need to notify a kernel queue at secondary interrupt level.

For each simultaneously pending kernel queue entry your software will place in a kernel queue from secondary interrupt level, you should increment the kernel queue's SIH limit by one. For example, if your secondary interrupt handler notifies a kernel queue but doesn't do so again until the first notification has been processed, you only need to increment the kernel queue's SIH limit by one. If a kernel queue notification operation at secondary interrupt level would cause more entries to be pending in the kernel queue than specified by the kernel queue's SIH limit, the `NotifyKernelQueue` operation will fail with a `memFullErr`.

When you increment or decrement a kernel queue SIH limit, the new value of the limit is returned.

```
OSStatus AdjustKernelQueueSIHLimit (KernelQueueID theQueueID,
                                     long amount,
                                     unsigned long * newLimit);
```

Waiting on a Kernel Queue

```
OSStatus WaitOnKernelQueue (KernelQueueID theQueue,
                             void ** p1,
                             void ** p2,
                             void ** p3,
                             Duration timeout);
```

Kernel queues can be waited upon by specifying the ID of the kernel queue, locations for the queue data that the task is interested in, and a time limit to wait. If data is already queued, the first set of available words are returned to the waiter. Otherwise, the task waits on the kernel queue, relinquishing its execution time to other tasks in the system. As with event groups, if the time limit is exceeded, the task is made executable without any queue data being returned, and a `kernelIncompleteErr` is returned.

`theQueue` specifies the targeted Kernel Queue.

`p1` returns the first word of user-specified data.

`p2` returns the second word of user-specified data.

`p3` returns the third word of user-specified data.

timeout specifies how long to wait for the queue to be notified.

WaitOnKernelQueue returns:

- noErr if data was successfully delivered.

- kernelIncompleteErr if the time out was exceeded, or if the queue was deleted.

- Other microkernel errors may be returned.

KERNEL NOTIFICATION

Many services provided by the microkernel can take place in parallel with the execution of the task that requests the service. These services are said to be *asynchronous*. The microkernel supports three mechanisms for indicating the completion of an asynchronous request: event flags, kernel queues, and software interrupts.

Asynchronous services allow you to specify a KernelNotification record that governs how you'll be informed of the request's completion. A kernel notification allows you to select any or all of the notification schemes. Kernel notifications are used in conjunction with various address space, timer, task, and message operations.

When an asynchronous kernel service completes, those services *deliver* the notification. Notification delivery is defined as the following actions in order:

- Setting one or more event flags within a specific event group.
- Sending a specific software interrupt. The service's result is used as the value of the second parameter to the software interrupt handler.
- Notifying a kernel queue. The p1 and p2 parameters are delivered as specified in the KernelNotification record, and p3 is set to the service's result.

Delivery of a notification is completely asynchronous to the execution of the task that is being notified.

Every microkernel service that makes use of the kernel notification mechanism has a notification parameter. This parameter is the address of a kernel notification record. If you pass a nil address, no notification will be delivered. Although kernel notification records are passed by address, the microkernel makes a complete copy of the record that you supply at the time that you call the service. The record that you supply is not referenced at the time the service completes and the notification is delivered.

Kernel Notification

Below is the type declaration for KernelNotification.

```
typedef struct KernelNotification
{
    EventGroupID      eventGroup;
    EventMask         eventMask;
}
```

```
    SoftwareInterruptId  swi;  
    KernelQueueID       kernelQueue;  
    void *              p1;  
    void *              p2;  
} KernelNotification;
```

TIMING SERVICES

The microkernel's timing services provide four different kinds of timers as well as services that allow you to determine the timer accuracy of the hardware and to get the current time.

Three of the timer services are used when tasks need to delay for a period of time or receive notification at a particular time. The fourth timer service allows you to specify a secondary interrupt handler that is to be run at a particular time.

Timer Accuracy

The accuracy of timer operations is quite good. Every attempt is made to ensure the quality of timed operations. However, certain limitations are inherent in the timing mechanisms. These limitations are described in this section.

About the Time Base

Timer hardware within the system is clocked at a rate that is CPU dependent. This rate is called the *time base*. The timer services isolate you from the time base by representing all times in `AbsoluteTime`. You can use the provided conversion routines to convert from `Nanoseconds` or `Duration` into `AbsoluteTime` system units. The times that you specify must be converted into the microkernel unit of `AbsoluteTime` to use the microkernel's timer services. This conversion can introduce errors. These errors are typically limited to one unit of the time base.

When performing sensitive timing operations, it can be important to know the underlying time base. For example, if the time base is 10 milliseconds, there is not much value in setting timers for 1 millisecond. You can determine the hardware time base by using the following service:

```
void          GetTimeBaseInfo (UInt32 *   theMinAbsoluteTimeDelta,
                               UInt32 *   theAbsTimeToNsecNumerator,
                               UInt32 *   theAbsTimeToNsecDenominator,
                               UInt32 *   theCPUToAbsTimeNumerator,
                               UInt32 *   theCPUToAbsTimeDenominator);
```

Representing the time base is difficult. The value is typically an irrational number. The microkernel solves this problem by returning a representation of the time base in fractional form: two 32-bit integer values, a numerator and denominator. The result of dividing the numerator by the denominator is a value that is equal to the number of `AbsoluteTime` units per `Nanoseconds`. If you

multiply an `AbsoluteTime` by the `conversionNumerator` divided by the `conversionDenominator`, the result will be in Nanoseconds.

The `minAbsoluteTimeDelta` is the minimum number of `AbsoluteTime` units that can change at any given time. For example, if the hardware increments the decremter in units of 128, then the `minAbsoluteTimeDelta` returned by `TimeBaseInfo` would be 128.

Setting the Time Base

The time base information is configurable for changes in the environment, such as the processor going to sleep.

Note: `SetTimeBaseInfo` must be called with interrupts disabled.

This is to insure that no clients of the timing services will be given feedback based on partially set timing information.

The original time base information can be obtained by calling `GetTimeBaseInfo`. To change the time base information, disable interrupts and call `SetTimeBaseInfo` with the new information. After the call completes, with interrupts still disabled, make the change to the timing hardware. Then re-enable interrupts.

Keep in mind that the smaller the conversion numerator and denominator, the better. The conversions are done by multiplying by the numerator and then dividing that result by the denominator. This means that smaller conversion numbers will allow for better use of the 64-bit `AbsoluteTime` without overflow.

```
void SetTimeBaseInfo (UInt32      newMinAbsoluteTimeDelta,  
                    UInt32      newConversionNumerator,  
                    UInt32      newConversionDenominator);
```

Timing Latency

Timing latency is the amount of time that passes between the expiration of a timer and receipt of notification that the timer has expired. Timing latency within the system is not deterministic. The effects of scheduling operations triggered by timer expiration do not necessarily occur immediately. Hardware interrupt handlers, secondary interrupt handlers, and tasks of greater or equal CPU priority will all contribute to the perceived latency of these timing services. Latency, by its nature, is not constant over time. Under some conditions, such as servicing a page fault when invoking the timer handler, latency may be larger

than the requested time interval. If you avoid installing many timers that all expire at nearly the same time, timer latency should be acceptable.

Timer Overhead

When setting a timer, the time you specify is used directly to program the timing hardware. The microkernel does not attempt to account for either the overhead of setting up the timer or the overhead of notifying a client of the timer's completion. As a result, a timed operation of one millisecond may actually require, for example, 1.1 milliseconds. Overhead is different from latency because it is constant.

Obtaining the Time

You can read the internal representation of time to which all timer services are referenced. This value starts at zero during microkernel initialization and increases throughout the system's lifetime. UpTime is currently located in the Timing shared library.

```
AbsoluteTime UpTime (void);
```

Setting Timers to Expire in the Past

Several of the timer services allow you to specify an absolute time at which the timer is to expire. It is, therefore, possible that the time you specify has already occurred. The microkernel does not attempt to optimize these cases. Timers set at times in the past will expire within a very short period of time, perhaps instantly, perhaps not. You should not depend upon the exact behavior of such timers.

Synchronous Timers

Synchronous timers cause the calling task to stop executing until a specific time is reached. The microkernel provides synchronous timers that specify time in both absolute and relative terms.

Synchronous Timers With Absolute Times

```
void DelayUntil (AbsoluteTime * expirationTime);
```

The calling task is blocked until expirationTime, at which point the task is again made eligible for execution. Unless called with software interrupts disabled or from within a software interrupt handler, software interrupts may be received while the task is otherwise sleeping. Because DelayUntil allows you to specify an absolute expiration time, you can perform periodic work at intervals that do not have long term drift.

Here is an example of a task that performs some work at one second intervals with no long term drift:

```
OSStatus DriftFreeWorker (void * work)
{
    TimeFormat  nextWorkTime;

    UpTime (&nextWorkTime);           // Get the time reference
    do
    {
        DoTheWork      (work)           // Do the work
        AddOneSecond   (&nextWorkTime); // Calculate next time to work
        DelayUntil     (&nextWorkTime); // Delay until that time
    } while (true);
}
```

Synchronous Timers With Relative Times

```
void DelayFor (Duration  theDelay);
```

The calling task is blocked for the amount of time specified. DelayFor allows the caller to delay for a time relative to when the service is called. You cannot achieve drift-free timing by using repeated calls to DelayFor.

Here is an example of a task that performs some work at one second intervals with unpredictable long term drift:

```
OSStatus DriftingWorker (void * work)
{
    do
    {
        DoTheWork      (work);           // Do the work
        DelayFor (durationSecond);       // Delay for one second
    } while (true);
}
```

DelayFor (0) relinquishes the CPU without blocking the caller. It lets other tasks at the same priority run. When all other tasks at the same priority have blocked or used their time slice, the caller will run again. If there are no other tasks at the same priority, DelayFor (0) has no effect.

Asynchronous Timers

Asynchronous timers cause the task to be notified when a specified time is reached. Starting an asynchronous timer yields an ID that you can use to cancel the timer prior to its expiration. Notification of timer expiration is done through the kernel notification mechanism.

```
typedef OptionBits TimerOptions;

OSStatus SetTimer (AbsoluteTime * expirationTime,
                  KernelNotification * notification,
                  TimerOptions options,
                  TimerID * theTimer);
```

A timer is set and upon expiration a notification is delivered.

expirationTime specifies the absolute time at which the notification should be generated.

notification specifies the manner in which the caller wishes to be notified upon expiration of the timer. For additional details, see the section “Kernel Notification,” earlier in this document .

options is currently unused.

theTimer is updated to reflect the ID of the created timer. This ID can only be used to cancel the timer prior to its expiration. The TimerID becomes invalid when either a CancelTimer operation is performed or the timer expires.

If notification is set to deliver a software interrupt, the p2 parameter of the delivered software interrupt provides special information. If the address space when the timer expires is the same as for the task that set the timer, the p2 parameter is set to the current PC at the time the timer expired. Otherwise, the p2 parameter is set to 0. This can be used for adaptive sampling.

If a kernel queue notification is specified, the p3 kernel queue parameter is set the same way the p2 parameter is set for a software interrupt. It is not recommended that queues be used for high-frequency sampling because doing so would require a large amount of microkernel resources.

Resetting Asynchronous Timers

An asynchronous timer that has been started, but has not yet expired or been canceled, can be reset. The reset operation can specify a new expiration time, a

new notification parameters, or both. If the timer has already expired, the ResetTimer call will fail with a kernelIDerr status.

```
OSStatus ResetTimer (TimerID          theTimer
                    AbsoluteTime *    expirationTime,
                    KernelNotification * notification);
```

An existing timer's expiration time and/or notification parameters are reset.

theTimer is the ID of the indicated timer.

expirationTime specifies the new absolute time at which the notification should be generated. If expirationTime is nil, the expiration time is not changed.

notification specifies the manner in which the caller wishes to be notified upon expiration of the timer. For additional details, see the section "Asynchronous Timers," earlier in this document. If notification is nil, the notification parameters of the timer are not changed.

Interrupt Timers

Each of the timing operations previously discussed are only pertinent to task-level execution. Certain device drivers and other low level software may require timers that have less latency or that can be set from hardware interrupt handlers. Interrupt timers fulfill both of these requirements.

Interrupt timers allow you to specify a secondary interrupt handler that is to be run when the timer expires. They are asynchronous in nature. You can set an interrupt timer from a hardware interrupt handler, a secondary interrupt handler, or a privileged task. Because there is no way to determine the current address space ID when the timer interrupt handler is called, interrupt timers are primarily useful for profiling privileged software and native user mode code that is executed from a mapped file. Emulated code will always result in PC values inside the emulator, and code stored in resources will result in a PC that is only valid in a particular address space — but there is no way to tell which address space.

Interrupt timers require the use of preallocated microkernel resources. A finite number of these timers are available, so they should be used only when no alternative exists.

```
OSStatus SetInterruptTimer(AbsoluteTime *    expirationTime,
                          SecondaryInterruptHandler handler,
                          void *           pl,
                          TimerID *       theTimer);
```

expirationTime is the absolute time at which the timer is to expire.

handler is the address of a secondary interrupt handler that is to be run when the specified time is reached.

p1 is the value that will be passed as the first parameter to the secondary interrupt handler when the timer expires. The value of the second parameter passed to the secondary interrupt handler is set to the current PC at the time the timer expired.

theTimer is updated with the ID of the timer that is created. This ID may be used in conjunction with CancelTimer.

Interrupt timers consume microkernel resources from the time they expire until the time they begin to execute. Since the microkernel can't allocate these resources dynamically at interrupt level, you must call AdjustInterruptTimerSIHLimit at task level to allocate them. You should do this once during your software's initialization, not each time you need to set an interrupt timer.

For each simultaneously active interrupt timer you use, you should increment the interrupt timer limit by one. For example, if you set an interrupt timer and don't set any additional interrupt timers the first one has run, you only need to increment the interrupt timer limit by one. When your software terminates or is removed, it should decrement the interrupt timer limit.

Note: If you fail to increment the interrupt timer limit, your software may appear to work fine. However, if an attempt is made to set more interrupt timers than allowed by the limit, a SetInterruptTimer call may fail with memFullErr. Because the limit manipulated by AdjustInterruptTimerSIHLimit is shared by all callers of SetInterruptTimer, the software that receives a bad status from SetInterruptTimer will not necessarily be the software that failed to call AdjustInterruptTimerSIHLimit.

When you increment or decrement the interrupt timer limit, the new value of the limit is returned.

```
OSStatus AdjustInterruptTimerSiHLimit
    (long amount,
     unsigned long * newLimit);
```

Canceling Asynchronous and Interrupt Timers

Canceling an outstanding asynchronous timer prevents the notification from being delivered. When you attempt to cancel an asynchronous timer, a race condition begins between your cancellation request and expiration of the timer. It is, therefore, possible that the timer will expire and that your cancellation attempt will fail even though the timer had not yet expired at the instant the cancellation attempt was made.

Attempts to cancel interrupt timers that are made at interrupt level are slightly less deterministic. The microkernel cannot cancel the actual timer until secondary interrupt time. So it is possible that the timer will expire and the secondary interrupt handler associated with the timer is run even though the timer was canceled. However, if the hardware interrupt handler that cancels the interrupt timer queues a secondary interrupt handler after it has made the cancellation request, the microkernel guarantees that the interrupt timer will have either run or been canceled before that secondary interrupt handler executes.

```
OSStatus CancelTimer (TimerID theTimer,
                     AbsoluteTime * timeRemaining);
```

CancelTimer cancels a previously created timer, and returns the amount of time remaining until the timer would have expired. An error is returned if the timer has either already expired or has been canceled.

ADDRESS SPACE MANAGEMENT

Address space management is the creation, deletion, and maintenance of logical address spaces. Address spaces are composed of memory ranges, called *areas*, that possess a set of common attributes including backing store and protection level. Maintenance services include those for copying data between address spaces and for controlling access to and paging of areas within a particular address space.

Commonalities in address space management services are noted in the following list. Deviations are mentioned in the descriptions of individual services.

- The kernelIDerr error is returned when the specified address space or area does not exist.
- When a logical address range (base and length) is specified, that range must lie entirely within a single area. Although some range operations need to be implemented in a page-aligned fashion, it is never required that the base and length be specified page-aligned. Further, these calls require that area be based in RAM, not in ROM or I/O space.

Basic Types

This section defines some types and values that are fundamental to address space management. The significance of the items mentioned is clarified by the descriptions of the services that use them.

Values of type LogicalAddress represent location in an address space or area.

```
typedef void * LogicalAddress;
```

Values of type PhysicalAddress represent location in physical memory. They are used primarily with backing object and DMA I/O operations.

```
typedef void * PhysicalAddress;
```

Address spaces are referred to by values of type AddressSpaceID. The function CurrentAddressSpace() returns the AddressSpaceID for the current address space.

Values of type MemoryAccessLevel represent allowable accesses to some portion of memory.

```
typedef UInt32 MemoryAccessLevel;  
enum
```

```

{
    kMemoryExcluded      = 0,
    kMemoryReadOnly     = 1,
    kMemoryReadWrite    = 2,
    kMemoryCopyOnWrite  = 3
};

```

- kMemoryExcluded specifies that no accesses at all, including instruction fetches, are allowed.
- kMemoryReadOnly specifies that read and instruction fetch operations are allowed.
- kMemoryReadWrite specifies that read, write and instruction fetch operations are allowed.
- kMemoryCopyOnWrite specifies that read, write and instruction fetch operations are allowed, but that modifying data in the area does not alter data in the backing store. Note that there is no way to revert modified copy-on-write pages to their original state.

```
typedef UnsignedWide BackingAddress;
```

Values of type BackingAddress are used to specify offsets within backing objects. They are 64-bit integer values in anticipation of file systems that provide support for files larger than 4 GB.

Static Logical Addresses

It is sometimes necessary to access the physical pages through logical addresses regardless of whether the physical page is mapped into the current address space. To enable this capability, the memory system keeps a *static mapping* of physical pages such that physical pages are mapped into a globally shared area at all times but are accessible only in privileged mode. The logical address in the static mapping corresponding to a given physical address is called the *static logical address* of the page. Static logical addresses are kept in variables of type LogicalAddress. They are valid for access by privileged software only.

Address Space Control

The following services support the creation and deletion of address spaces. Others allow the caller to obtain information about the address spaces already in existence.

Creating Address Spaces

Address spaces can be created to provide additional addressing and protection.

```
OSStatus CreateAddressSpace (AddressSpaceID * theAddressSpace);
```

CreateAddressSpace builds a new address space and returns an AddressSpaceID for it. A new address space automatically contains any existing global areas and memory reservations. (For details, see the sections “Creating Areas” and “Creating Memory Reservations,” respectively, later in this document.)

theAddressSpace is an output parameter indicating the address space identifier that can be used for subsequent operations on the created address space. A value of kInvalidID will be returned if CreateAddressSpace fails.

Deleting Address Spaces

```
OSStatus DeleteAddressSpace (AddressSpaceID theAddressSpace);
```

DeleteAddressSpace destroys the specified address space. All non-global areas mapped into that space are also destroyed.

Note: Care should be taken to prevent references to the deleted address space.

theAddressSpace specifies the address space to destroy.

Obtaining Information About an Address Space

```
struct SpaceInformation
{
    ItemCount    numLogicalPages;
    ItemCount    numInMemoryPages;
    ItemCount    numResidentPages;
};
typedef struct SpaceInformation SpaceInformation;
```

```
enum
{
    kSpaceInformationVersion = 1
};
```

```
OSStatus GetSpaceInformation (AddressSpaceID theAddressSpace,
                             PBVersion       theVersion,
                             SpaceInformation * theSpaceInfo);
```

GetSpaceInformation returns information about the specified address space.

theAddressSpace specifies the address space for which to get the information.

theVersion specifies the version number of SpaceInformation to be returned. This parameter will provide backward compatibility. kSpaceInformationVersion is the version of SpaceInformation defined in the current interface.

theSpaceInfo specifies where to return the information.

The fields of a SpaceInformation structure are:

- identity - the AddressSpaceID of the address space.
- numLogicalPages -the total number of logical pages in this address space (doesn't include pages in global areas)
- numInMemoryPages - the number of logical pages in this address space currently in memory (doesn't include pages in global areas).
- numResidentPages - the number of pages in this address space locked in physical memory (includes resident areas, pages locked with ControlPagingForRange, and pages locked with PrepareMemoryForIO).

```
AddressSpaceID CurrentAddressSpaceID (void);
```

CurrentAddressSpaceID returns the ID of the current address space.

Iterating Over All Address Spaces

Tasks can obtain the AddressSpaceIDs of all the existing address spaces.

```
OSStatus GetAddressSpacesInSystem  
        (ItemCount          requestedAddressSpaces,  
         ItemCount *       totalAddressSpaces,  
         AddressSpaceID *  theAddressSpaces);
```

GetAddressSpacesInSystem returns the IDs of all the address spaces within the system.

requestedAddressSpaces indicates the maximum number of address space IDs that should be returned (that is,. the number of entries available at the location pointed to by theAddressSpaces.)

totalAddressSpaces is filled in with the total number of address spaces in the system. If less than or equal to requestedAddressSpaces, all address spaces were returned; if greater than requestedAddressSpaces, insufficient space was available to return all AddressSpaceIDs.

theAddressSpaces is filled in with the IDs of the address spaces in the system.

Area Control

The following operations provide support for creating and deleting areas within address spaces. Other operations allow the caller to obtain information about the areas already in existence.

Creating Areas

```
typedef OptionBits AreaOptions;
enum
{
    kZeroFill          = 0x00000001,
    kResidentArea      = 0x00000002,
    kSparseArea        = 0x00000004,
    kPlacedArea        = 0x00000008,
    kGlobalArea        = 0x00000010
};
```

```
OSStatus CreateArea (KernelProcessID    owningKernelProcess,
                    BackingObjectID     backingObject,
                    BackingAddress *    backingBase,
                    ByteCount            backingLength,
                    MemoryAccessLevel    userAccessLevel,
                    MemoryAccessLevel    privilegedAccessLevel,
                    ByteCount            guardLength,
                    AreaOptions          options,
                    LogicalAddress *     areaBase,
                    AreaID *             theArea);
```

CreateArea creates a mapping between the specified address space and the specified backing store. The AreaID of the newly created area and the logical address of that area's origin are both returned to the caller. The logical address has meaning only within the context of the area's owning address space.

owningKernelProcess specifies the kernel process in whose address space to create the area. Areas are automatically reclaimed when their owning kernel processes are deleted.

backingObject specifies the backing store whose content is to be mapped. Specifying kNoBackingObject for this parameter implies that a scratch backing store file should be used. If either the kResidentArea option is specified, or if all access to the area is excluded, backingObject must be kNoBackingObject.

backingBase specifies the offset within backingObject that is to correspond to the lowest address in the area. Note that this parameter is the address of the actual BackingAddress parameter. backingBase being nil specifies a BackingAddress of zero. The range of possible BackingAddress values is not constrained by the memory system. Backing objects themselves may place restrictions (e.g. on a block-oriented device, the base might need to be a whole multiple of the block size). If the kResidentArea option is specified, backingBase must be specified as nil.

backingLength specifies the number of bytes to map from backingObject, starting at backingBase. It will be rounded up to a multiple of the logical page size. This implies that more backing store than was specified may be mapped in. backingLength must be non-zero.

userAccessLevel and privilegedAccessLevel specify the kinds of memory references that non-privileged and privileged software are allowed to make in the area, respectively. References made in violation of the access level result in exceptions at the time of the access. See the section “Memory Exceptions,” later in this document. If privilegedAccessLevel is more restrictive than userAccessLevel, privilegedAccessLevel will be made equal to userAccessLevel. kMemoryCopyOnWrite is not allowed for resident areas. Note that to create a truly excluded area, the both access levels should be kMemoryExcluded, and both the kSparseArea and kResidentArea AreaOptions should be specified. If those options are not specified, physical memory and /or disk space will be assigned to the area. Area access levels can be adjusted by the SetAreaAccess service.

guardLength specifies the size, in bytes, of the excluded logical address ranges to place adjacent to each end of the area. The ranges, called *area guards*, are excluded to both privileged and non-privileged software. References to those addresses result in exceptions. For details, see the section “Memory Exceptions,” later in this document. The guardLength will be page-aligned, if necessary. This means that the excluded ranges may be larger than is specified.

options specifies desired characteristics of the area being created. Values for this parameter are defined by the AreaOptions type.

- kZeroFill specifies that memory in this area should be initialized to zero. This option applies only to scratch areas (i.e., kNoBackingObjectID is specified in backingObject) and non-pageable areas (i.e. the kResidentArea option is specified).
- kResidentArea specifies that the data for this area must always be physically resident. These areas are never paged between memory and backing storage. This option is available only to privileged callers.

- `kSparseArea` specifies that the resources for the area be allocated on-demand. This option applies only to scratch areas (i.e., `kNoBackingObjectID` is specified in `backingObject`) and non-pageable areas (i.e. the `kResidentArea` option is specified). For scratch areas, sparseness means that the scratch backing object will be sparse, if possible. For resident areas, sparseness means that the physical memory will be allocated by page faulting.
- `kPlacedArea` specifies that `areaBase` specifies where to create the area. `areaBase` and `backingLength` will be page-aligned. This means that the area may be larger than was specified. `CreateArea` fails and an error is returned if the area cannot be so positioned. `areaBase` will be set to the actual beginning of the area.

Note: Care should be taken when using the `kPlacedArea` option, as the specified location might be part of a memory reservation unknown to the caller. It is advisable to create a reservation for the range in which the area will be placed, prior to creating the area. Reservations can be made either for a specific address space, or globally. For details, see the section “Creating Memory Reservations,” later in this document.

- `kGlobalArea` specifies that the data for this area is to be addressable from any address space. All address spaces will get access in accordance with the privileged and non-privileged access levels specified. The created area appears at `areaBase` in every address space.
- `kPhysicallyContiguousArea` can only be specified if `kResidentArea` is also specified. `kPhysicallyContiguousArea` specifies that the physical pages that make up the area must be contiguous. This option should only be used by device drivers for devices that must perform multi-page DMA transfers but do not handle scatter/gather operations. A `CreateArea` call with the `kPhysicallyContiguousArea` option may fail even when there is a great deal of available memory.

`areaBase` is an output parameter indicating the beginning logical address of the mapped memory. If the `kPlacedArea` option is specified, `areaBase` is also an input specifying where to position the area. See the description of `kPlacedArea`, earlier in this section.

`theArea` is an output parameter indicating the area identifier that can be used for subsequent operations on the created area. A value of `kInvalidID` will be returned if `CreateArea` fails.

Deleting Areas

```
OSStatus DeleteArea (AreaID theArea);
```

DeleteArea removes the specified area. If the area is global, it is deleted from all address spaces. Further references to the logical addresses previously mapped will result in memory exceptions. Non-global areas are also deleted if the address space containing them is deleted.

Note: DeleteArea has no formal interactions with other pieces of system software. Care should be taken to prevent potential references to the deleted area.

theArea specifies the area to destroy.

Obtaining Information About an Area

```
struct AreaInformation
{
    AddressSpaceID    addressSpace;
    LogicalAddress    base;
    ByteCount         length;
    MemoryAccessLevel userAccessLevel;
    MemoryAccessLevel privilegedAccessLevel;
    AreaUsage         usage;
    BackingObjectID   backingObject;
    BackingAddress    backingBase;
    AreaOptions       options;
    KernelProcessID   owningKernelProcess;
};
typedef struct AreaInformation    AreaInformation;

enum
{
    kAreaInformationVersion = 1
};

OSStatus GetAreaInformation (AreaID          theArea,
                             PBVersion      version,
                             AreaInformation * areaInfo);
```

GetAreaInformation returns information about the specified area.

theArea specifies the area for which to return information.

version specifies the version number of AreaInformation to be returned. This provides backwards compatibility. kAreaInformationVersion is the version of AreaInformation defined in the current interface.

areaInfo specifies where to return the information.

The fields of an AreaInformation structure are:

- addressSpace - the address space that contains the area.
- base - the logical address of the area.
- length - the size, in bytes, of the area.
- userAccessLevel - the kinds of references allowed by non-privileged execution.
- privilegedAccessLevel - the kinds of references allowed by privileged execution.
- usage - what the area is used for - RAM, I/O, onboard video, etc.
- backingObject - the object providing backing store for the area. The value noBackingObjectID is returned if there is no backing object .
- backingBase - the area's base address within the backingObject.
- options - the options that were specified at the time the area was created.
- owningKernelProcess - the KernelProcessID that was specified when the area was created.

For further information about access levels and area options, see the description of CreateArea in the section "Creating Areas," earlier in this document.

Iterating Over All Areas Within an Address Space

```
OSStatus GetAreasInAddressSpace (AddressSpaceID  addressSpace ,
                                itemCount        requestedAreas ,
                                itemCount *      totalAreas ,
                                AreaID *        theAreas ) ;
```

GetAreasInAddressSpace allows the caller to find the IDs of all tasks within a particular address space. For additional information about using iteration functions see the section "Some Basic Types," earlier in this document.

addressSpace indicates the address space of interest.

requestedAreas indicates the maximum number of area IDs that should be returned. This indicates the number of entries available at the location pointed to by theAreas.

totalAreas is filled in with the total number of areas within the address space. If less than or equal to requestedAreas, all areas were returned; if greater than requestedAreas, insufficient space was available to return all area IDs.

theAreas is filled in with the IDs of the areas within addressSpace.

The status return is:

- noErr if addressSpace exists, whether or not all area IDs were returned.
- kernelIDerr if addressSpace doesn't exist.

Changing the Access Level of an Area

It is sometimes useful to change the kind of accesses that are allowed to an area. For example, a code loader might need to make an area read-write while initializing it, then change it to read-only when the area is ready to use.

```
OSStatus SetAreaAccess (AreaID theArea,
                       MemoryAccessLevel userAccessLevel,
                       MemoryAccessLevel privilegedAccessLevel);
```

SetAreaAccess changes the allowed accesses to an area.

theArea specifies the AreaID of the area in which to change the access.

userAccessLevel and privilegedAccessLevel specify the kinds of memory references that non-privileged and privileged software are allowed to make in the area, respectively. A reference made in violation of the access level results in an exception at the time of the access. For details, see the section "Memory Exceptions," later in this document. If privilegedAccessLevel is more restrictive than userAccessLevel, privilegedAccessLevel will be made equal to userAccessLevel. kMemoryCopyOnWrite is not allowed for resident areas.

Finding the Area That Contains a Particular Logical Address

To find the area that contains a logical, call GetAreaFromAddress.

```
OSStatus GetAreaFromAddress (AddressSpaceID addressSpace,
                             LogicalAddress address,
                             AreaID * theArea);
```

GetAreaFromAddress returns the AreaID of the area associated with the specified logical address.

addressSpace specifies the address space containing the logical address in question.

address specifies the logical address to look up.

theArea is an output parameter where the AreaID of the logical address is returned.

Using Areas to Access Large Backing Stores

Some backing stores are too large to view in their entirety in the space available to a single address space. A common way to deal with this limitation is to create a limited-size mapping (area) and then adjust where in the backing store that mapping corresponds.

```
OSStatus SetAreaBackingBase (AreaID          theArea,
                             const BackingAddress * backingBase);
```

SetAreaBackingBase sets the specified BackingAddress as the base for the specified area. An area's base BackingAddress and length determine which portion of the backing object is mapped to the area. Changing an area's base BackingAddress is an effective method for accessing numerous parts of a large backing store through a relatively small logical address range.

theArea specifies the area in which to change the backing store base.

backingBase specifies the offset within the backing object that is to correspond to the lowest address in the area. Note that this parameter is the address of the actual BackingAddress parameter. A backingBase that is nil specifies a BackingAddress of zero. The range of possible BackingAddress values is not constrained by the memory system. Backing objects themselves may place restrictions (e.g. on a block-oriented device, the base might need to be a whole multiple of the block size).

Memory Control

Obtaining Information About a Range of Logical Memory

You can obtain usage information for each logical page within a range of logical addresses. This information may be useful when performing certain runtime operations such as garbage collection and/or heap compaction.

```
enum
{
    kPageInformationVersion = 1
};

typedef UInt32      PageStateInformation;
enum
```

```

{
    kPageIsProtected          = 0x00000001,
    kPageIsProtectedPrivileged = 0x00000002,
    kPageIsModified          = 0x00000004,
    kPageIsReferenced        = 0x00000008,
    kPageIsLockedResident    = 0x00000010,
    kPageIsInMemory          = 0x00000020,
    kPageIsShared            = 0x00000040,
    kPageIsWriteThroughCached = 0x00000080,
    kPageIsCopyBackCached    = 0x00000100
};

struct PageInformation
{
    AreaID          area;
    ItemCount       count;
    PageStateInformation information [1];
};
typedef struct PageInformation PageInformation;

OSStatus GetPageInformation (AddressSpaceID addressSpace,
                            LogicalAddress base,
                            ItemCount requestedPages,
                            PBVersion version,
                            PageInformation * thePageInfo);

```

GetPageInformation returns information about each logical page in the specified range.

addressSpace specifies the address space containing the range of interest.

base is the first logical address of interest.

requestedPages specifies the number of pages for which information is to be returned.

version specifies the version number of PageInformation to be returned. This provides backwards compatibility. kPageInformationVersion is the version of PageInformation defined in the current interface.

thePageInfo is filled in with information about each logical page. This buffer has space for requestedPages entries. Page information is as follows:

- area indicates the AreaID of the area associated with the range.
- count indicates the number of entries in which information was returned.
- information contains one PageStateInformation entry for each logical page.

The bits of PageStateInformation are as follows:

- `kPageIsProtected` - the page is write protected against unprivileged software.
- `kPageIsProtectedPrivileged` - the page is write protected against privileged software.
- `kPageIsModified` - the page has been modified since the last time it was mapped in or its data was released (by the `ReleaseData` service).
- `kPageIsReferenced` - the page has been referenced (either load or store) since the last time the memory system's aging operation checked the page.
- `kPageIsLockedResident` - the page is ineligible for replacement (i.e. non-pageable) because there is at least one outstanding `PrepareMemoryForIO` and/or `SetPagingMode` (of `kPagingModeResident`) against it.
- `kPageIsInMemory` - the page is present in physical memory.
- `kPageIsShared` - the page's underlying physical page is mapped into additional logical pages.
- `kPageIsWriteThroughCached` - modifications to the page are written through the processor cache to main memory.
- `kPageIsCopyBackCached` - modifications to the page may be cached by the processor and not immediately reflected in main memory.

Data to Code

Placing executable data in memory requires synchronization with the processor's data and instruction caches. The details are specific to the processor and the internal operation of the memory system. Consequently, the memory system provides services that encapsulate the necessary operations.

```
OSStatus DataToCode (AddressSpaceID addressSpace,
                    LogicalAddress base,
                    ByteCount      length);
```

`DataToCode` performs the operations necessary for the specified memory range to be treated as processor instructions instead of simple data. This is required, for example, when reading instructions into scratch memory, or when generating instructions "on the fly."

`addressSpace` specifies the address space containing the range to be treated as code.

`base` specifies the start of the range to be treated as code.

`length` specifies the number of bytes in the range to be treated as code.

The beginning and end of the range will be adjusted, if necessary, so that the range begins and ends on logical page boundaries. This means that more memory than was specified may be affected.

Controlling Memory Cacheability

```
typedef UInt32 ProcessorCacheMode;
enum {
    kProcessorCacheModeDefault          = 0,
    kProcessorCacheModeInhibited        = 1,
    kProcessorCacheModeWriteThrough     = 2,
    kProcessorCacheModeCopyBack         = 3
};

OSStatus      SetProcessorCacheMode
               (AddressSpaceID      addressSpace,
                LogicalAddress       base,
                ByteCount            length,
                ProcessorCacheMode    processorCacheMode);
```

SetProcessorCacheMode sets the memory hardware cache mode for the specified range.

addressSpace specifies the address space containing the range to change.

base specifies the start of the range.

length specifies the number of bytes in the range.

processorCacheMode specifies the cache mode. Values for this parameter are defined by the ProcessorCacheMode type.

- kProcessorCacheModeDefault specifies the cache mode inherent to the range.
- kProcessorCacheModeInhibited specifies that data and/or code caching are to be disabled.
- kProcessorCacheModeWriteThrough specifies that memory read operations utilize the cache, but that the effect of write operations is immediately apparent in physical memory.
- kProcessorCacheModeCopyBack specifies that both read and write operations utilize the cache (implying that the effect of write operations need not be immediately apparent in physical memory).

The beginning and end of the range will be adjusted, if necessary, so that the range committed begins and ends on logical page boundaries. This means that *more* memory than was specified may be affected.

Controlling Paging Operations

Paging operations on a memory range can be altered programmatically by using `ControlPagingForRange`.

```
typedef UInt32    PageControlOperation;
enum
{
    kControlPageMakePageable    = 1,
    kControlPageMakeResident    = 2,
    kControlPageCommit          = 3,
    kControlPageTouch           = 4,
    kControlPageReplace         = 5
};

OSStatus ControlPagingForRange (AddressSpaceID    addressSpace,
                               LogicalAddress     base,
                               ByteCount         length,
                               PageControlOperation operation);
```

`ControlPagingForRange` controls paging operations on the specified range.

`addressSpace` specifies the address space containing the range to change.

`base` specifies the start of the range to change.

`length` specifies the number of bytes in the range to change.

The beginning and end of the range will be adjusted, if necessary, so that the range begins and ends on logical page boundaries. This means that more memory than was specified may be affected.

`operation` specifies the paging operation. Values for this parameter are defined by the `PageControlOperation` type.

- `kControlPageMakePageable` specifies to undo the effect of one call to set the page to `kControlPageMakeResident`. When all such calls have been undone, the page is returned to the mode associated with the area containing the range (i.e. pages in backed areas become eligible for replacement, but pages in resident areas do not).
- `kControlPageMakeResident` specifies that the range is to be loaded and made ineligible for page replacement.
- `kControlPageCommit` specifies that backing store is allocated for a page in a sparse pageable area, or physical memory is allocated for a page in a sparse resident area. Ranges in areas created without the `kSparseArea` option specified do not benefit from this operation (though `CommitRange` completes with a success status).

- `kControlPageTouch` specifies that the page is to be brought into physical memory. The caller is not blocked while this is done. This operation can be used to optimize performance by causing the system to read in pages that will be needed in the future.
- `kControlPageReplace` specifies that the physical memory space occupied by the page is to be made available for other uses, after writing the page data to backing store if necessary. The data in the page is always preserved by this operation. This operation can be used to optimize performance by giving the memory system information about pages that won't be needed in the near future.

A program making sequential references to an area of memory could use `ControlPagingForRange` to decrease the time it spends waiting for paging operations and the amount of memory it uses. Such a program would use the `kControlPageTouch` operation for addresses it is about to reference, and the `kControlPageReplace` operation for addresses it has finished referencing.

Preventing Unnecessary Backing Store Activity

```
typedef OptionBits ReleaseDataOptions
enum
{
    kReleaseBackingStore = 0x00000001
};

OSStatus      ReleaseData (AddressSpaceID      addressSpace,
                           LogicalAddress      base,
                           ByteCount           length,
                           ReleaseDataOptions  options);
```

`ReleaseData` informs the memory system that the data values in the specified range are no longer needed. It is an optimizing hint to prevent writing the data to the backing store. The backing store, if any, remains allocated to the range. This is useful, for example, when deallocating dirty heap blocks.

Note: If the released range is subsequently accessed, the values in memory will be unpredictable. This includes data in areas with `kMemoryCopyOnWrite` access: that is, the data is not guaranteed to revert to its original, unmodified, state.

`addressSpace` specifies the address space containing the range to release.

`base` specifies the start of the range to release.

`length` specifies the number of bytes in the range to release.

`options` values are specified by the `ReleaseDataOptions` type.

- `kReleaseBackingStore` causes the backing store associated with the range to be deallocated if appropriate. (For instance, backing store won't be deallocated if the backing object was opened with read only access.) Specifying this option frees backing store space but increases the runtime cost of the operation—and possibly incurs future costs when the page is touched again and backing store must be reallocated.

The beginning and end of the range will be adjusted, if necessary, so that the range released begins and ends on logical page boundaries. This means that *less* memory than was specified may be released.

Memory Control in Association With I/O Operations

Memory usage in a demand paged, multi-tasking system is both highly dynamic and highly complex. It follows that data transfers to and from memory require close cooperation with the memory system to ensure proper operation.

The first consideration is that physical memory must remain assigned to the I/O buffer for at least the duration of the transfer (and, for logical I/O operations, that memory accesses do not page fault).

The second consideration is memory coherency. On output, it is essential that data in the processor's data cache be included in the transfer. On input, it is essential that the caches (both data and code) not be left with any out-of-date information, and it is desirable if the data cache can contain at least some of the data that was transferred. Furthermore, cache architectures vary from processor to processor, so it is also desirable to minimize or eliminate processor dependencies in the I/O drivers.

The microkernel provides I/O support services that, when used properly, ensure that these considerations are met.

The most common I/O transaction expected is a one-shot transfer where the I/O buffer belongs to the driver's client, such as handling a page fault by reading data directly into the user's page. The design also allows for multiple transactions to occur on a single buffer. An example of this is a network driver whose transactions consist of reading data into its own buffer, processing the data, and then copying the data off to a client's buffer. In this case, the driver re-uses the same buffer for an indefinite number of transactions.

The two services the microkernel provides are `PrepareMemoryForIO` and `CheckpointIO`. `PrepareMemoryForIO` informs the microkernel that a particular buffer will be used for I/O transfers. It assigns physical memory to the buffer

and, optionally, prepares the processor's caches for a transfer. CheckpointIO informs the microkernel that the previously started transfer, if any, is complete, whether there will be more transfers, and optionally the direction of the next transfer. It finalizes the caches and, if the next I/O direction is specified, prepares the caches for that transfer. If its parameters specify that no more transfers will be made, CheckpointIO deallocates the microkernel resources associated with the buffer preparation: subsequent I/O operations on this range of memory will need to begin with a call to PrepareMemoryForIO.

In the one-shot scenario, a PrepareMemoryForIO call prior to the transfer and a single CheckpointIO call following the transfer are used. The PrepareMemoryForIO parameters would specify the buffer location and the I/O direction, the CheckpointIO parameters would specify that no more transfers will be made.

In the multiple transfer scenario, a PrepareMemoryForIO when the buffer is allocated, a CheckpointIO prior to each transfer, and a CheckpointIO when the buffer is deallocated are used. The PrepareMemoryForIO parameters would specify the buffer location, but might or might not specify the I/O direction. The I/O direction is omitted if the transfer is not imminent, because the cache preparation would be wasted. The CheckpointIO calls before each transfer would specify the direction of the transfer and that more transfers will be made (not needed before the first transfer, if the PrepareMemoryForIO parameters specified an I/O direction). The final CheckpointIO parameters would specify that no more transfers will be made.

Note: Failure to properly use these I/O-related microkernel services can result in data corruption and/or fatal system errors. Correct system behavior is the responsibility of the microkernel and all I/O components including drivers, managers, and hardware.

Note: The descriptions here are not, generally, sufficient to allow the reader to write a correct, high performance I/O driver. Guidelines for writing I/O drivers, including correct usage of the microkernel services dedicated to I/O support, are beyond the scope of this document.

Preparing For I/O

```
typedef OptionBits IOPreparationOptions;
enum {
    kIOMultipleRanges          = 0x00000001,
    kIOLogicalRanges          = 0x00000002,
    kIOMinimalLogicalMapping  = 0x00000004,
```

```

    kIOShareMappingTables      = 0x00000008,
    kIOIsInput                  = 0x00000010,
    kIOIsOutput                 = 0x00000020,
    kIOCoherentDataPath        = 0x00000040,
    kIOTransferIsLogical        = 0x00000080,
    kIOClientIsUserMode        = 0x00000080
};

typedef OptionBits IOPreparationState;
enum {
    kIOStateDone                = 0x00000001
};

typedef LogicalAddress *      LogicalMappingTablePtr;
typedef PhysicalAddress *    PhysicalMappingTablePtr;

struct AddressRange
{
    void *                    base;
    ByteCount                 length;
};
typedef struct AddressRange  AddressRange;
typedef struct AddressRange * AddressRangeTablePtr;

struct MultipleAddressRange
{
    ItemCount                 entryCount;
    AddressRangeTablePtr     rangeTable;
};
typedef struct MultipleAddressRange MultipleAddressRange;

struct IOPreparationTable
{
    IOPreparationOptions      options;
    IOPreparationState        state;
    IOPreparationID           preparationID;
    AddressSpaceID            addressSpace;
    ByteCount                 granularity;
    ByteCount                 firstPrepared;
    ByteCount                 lengthPrepared;
    ItemCount                 mappingEntryCount;
    LogicalMappingTablePtr    logicalMapping;
    PhysicalMappingTablePtr   physicalMapping;
    union
    {
        AddressRange          range;
        MultipleAddressRange  multipleRanges;
    }
};

typedef struct IOPreparationTable IOPreparationTable;

OSStatus PrepareMemoryForIO (IOPreparationTable* theIOPreparationTable);

```

PrepareMemoryForIO enables device input/output on one or more ranges to occur in a manner coordinated with the microkernel, the main processor caches and other data transfers. Preparation includes ensuring that physical memory is assigned, and remains assigned, to the range at least until CheckpointIO relinquishes it. Depending upon the I/O mode (programmed I/O and/or DMA), I/O direction and data path coherence that are specified, the microkernel manipulates the contents of the processor's caches, if any, and may make the underlying memory non-cacheable.

I/O preparation must be done prior to moving the data. For operations on block oriented devices, the preparation should be done just before moving the data, typically by the driver. For operations on buffers such as memory shared between the main processor and a co-processor, frame buffers, or buffers internal to a driver, the preparation should be performed when the buffer is allocated.

In the event that insufficient resources are available to prepare all of the specified memory, PrepareMemoryForIO will prepare as much as possible, and indicate to the caller which memory was prepared. This is referred to as a *partial preparation*. The caller can examine the kIOStateDone bit in the tableState to check whether the preparation completed or was partial, and can determine in either case which part of the overall range(s) was prepared by examining firstPrepared and lengthPrepared. See the descriptions of kIOStateDone, firstPrepared and lengthPrepared later in this section.

Memory must be prepared and finalized for the benefit of the system and other users of the memory and backing store, even if the caller does not need any of the information provided by PrepareMemoryForIO.

Calls to PrepareMemoryForIO should be matched with calls to CheckpointIO, even if the I/O was aborted. In addition to applying finishing operations to the memory range, CheckpointIO deallocates microkernel resources used in preparing the range.

theIOPreparationTable specifies the table that specifies the ranges to be prepared and provides storage for mapping information to be returned.

IOPreparationTable fields have the following meanings.

- options specifies optional characteristics of the IOPreparationTable and the transfer. This value contains bits with the following meanings:
 - kIOMultipleRanges specifies that the rangeInfo field is to be interpreted as a MultipleAddressRange, enabling a scatter-gather specification.

- `kIOLogicalRanges` specifies that the base fields of the `AddressRange` structures are logical addresses. If this option is omitted, the addresses are treated as physical addresses.
- `kIOMinimalLogicalMapping` specifies that the `LogicalMappingTable` is to be filled in with just the first and last mappings of each range, arranged as pairs. This is useful for transfers where physical addresses are used for the bulk of the transfer, but logical addresses must be used to handle unaligned portions at the beginning and end: it obviates allocating a full-sized `LogicalMappingTable`. Two entries per range are used, regardless of the range sizes. (However, the value of the second entry of the pair is undefined if the range is contained within a single page.)
- `IOShareMappingTables` specifies that the microkernel can use the caller's mapping tables rather than maintain its own copy of the tables. This option conserves microkernel resources but can be specified only if the mapping tables are located in logical memory that cannot page-fault, such as a resident area or a locked portion of a pageable area, and will remain so until the final `CheckpointIO` completes. The mapping tables must remain allocated and the entries unaltered until after the final `CheckpointIO`, as well. It is not necessary for the caller to provide both tables.
- `kIOIsInput` specifies that data will be moved into main memory.
- `kIOIsOutput` specifies that data will be moved out of main memory.

Note that `kIOIsInput` and `kIOIsOutput` are completely independent. You may specify either, both, or neither at preparation time. Specifying neither is useful when the preparation must be made long in advance of the transfer (i.e. so the system resources are allocated). `CheckpointIO` can then be called just prior to the transfer to prepare the caches.

- `kIOCoherentDataPath` indicates that the data path that will be used to access memory during the I/O operation is fully coherent with the main processor's data caches, obviating data cache manipulations. Coherency with the instruction cache is not implied, however, so the appropriate instruction cache manipulations are performed regardless. When in doubt, omit this option. Incorrectly omitting this option merely slows the operation of the computer, whereas incorrectly specifying this option can result in erroneous behavior and crashes.
- `kIOClientIsUserMode` indicates that `PrepareMemoryForIO` is being called on behalf of a non-privileged client. If this option is specified, the memory ranges are checked for user-mode accessibility. If this option is omitted, the memory ranges are checked for privileged-level accessibility. Drivers can obtain the client's execution mode via the Family Programming Interface (FPI). In general, however, this information is available to message recipients in the message header.
- state is filled in by `PrepareMemoryForIO` to indicate the state of the `IOPreparationTable`. `IOPreparationTableState` bits are:

- `kIOStateDone` indicates that `PrepareMemoryForIO` successfully prepared up to the end of the specified ranges. Regardless of whether this bit is set, `firstPrepared` and `lengthPrepared` indicate the range prepared.
- `preparationID` is filled in by `PrepareMemoryForIO` to indicate the identifier that represents the I/O transaction. When the I/O operation has been completed or aborted, this `IOPreparationID` is used to finish the transaction. For details, see the next section “Finalizing I/O.”
- `addressSpace` specifies the address space containing the logical ranges.
- `granularity` specifies a hint to `PrepareMemoryForIO` in the event of a partial preparation. It is useful for transfers with devices that operate on fixed-length buffers. The length prepared will be zero, or an integral multiple of `granularity` rounded up to the next greatest page alignment. This prevents preparing more memory than the caller is willing to use. A value of zero for `granularity` specifies no granularity. Note that there is no check for whether the specified range lengths are multiples of `granularity`.
- `firstPrepared` specifies the byte offset into the range(s) at which to begin preparation. Note that when a `MultipleAddressRange` is specified, this offset is into the aggregate range.
- `lengthPrepared` is filled in by `PrepareMemoryForIO` to indicate the number of bytes, starting at `firstPrepared`, that were prepared. This is true regardless of whether the preparation was partial or complete.

`firstPrepared` and `lengthPrepared` control partial preparations. When calling `PrepareMemoryForIO` the first time, specify zero for `firstPrepared`. If the `tableState` returned does not indicate `kIOStateDone`, a partial preparation was performed. After the transfer and final `CheckPointIO` are made against this preparation, another `PrepareMemoryForIO` call can be made to prepare as much as possible of the ranges that remain. This time, specify `firstPrepared` as the sum of the current `firstPrepared` and `lengthPrepared`. This sequence of prepare/transfer/final checkpoint can be repeated until completing the iteration when the `IOPreparationState` indicates `kIOStateDone`.

- `mappingEntryCount` specifies the number of entries in the `logicalMapping` and/or `physicalMapping` tables. Note, however, that the `logicalMapping` table is assumed to have two entries per range if the `kIOMinimalLogicalMapping` option is specified, regardless of `mappingEntryCount`. One entry per page is needed. If there are not enough entries, a partial preparation is performed within the limit of the tables and the `kIOStateDone` state bit is returned zero.
- `logicalMapping` specifies the address of the `LogicalMappingTable` to be filled in by `PrepareMemoryForIO`. The table needs to have as many entries as there are distinct logical pages in the range(s). This table is optional. A `nil` value specifies that there is no table.

On return, the LogicalMappingTable contains the static logical addresses corresponding to the ranges' physical addresses. The table is a concatenation, in order, of the mappings for each specified range. The first entry of each range's mappings will be the exact static logical mapping of the first prepared address in that range, regardless of page-alignment, while the remaining entries will be page-aligned. Alternatively, the kIOMinimalLogicalMapping option specifies to return just the first and last static logical mappings of each range.

Specifying a logical mapping table implies that the transfer will be made through the main processor's MMU and data caches. Such transfers are performed with devices that fall into the Programmed I/O category of I/O devices. PrepareMemoryForIO and CheckpointIO take this into account when determining which cache operations are needed before and after the transfer.

Note: PrepareMemoryForIO guarantees that the underlying physical memory remains assigned to the ranges at least until CheckpointIO relinquishes it. However, it does not guarantee that the original logical address ranges remain mapped. In particular, the controlling areas may be deleted before CheckpointIO. If the caller cannot somehow guarantee that the area(s) will continue to exist, logical address references to the underlying physical memory must be made through the static logical addresses provided in the mapping tables.

- physicalMapping specifies the address of the PhysicalMappingTable to be filled in by PrepareMemoryForIO. The table needs to have as many entries as there are distinct logical pages in the range. This table is optional. A nil value specifies that there is no table.

On return, the PhysicalMappingTable contains the physical addresses that comprise the ranges. The table is a concatenation, in order, of the mappings for each specified range. The first entry of each range's mappings will be the exact physical mapping of the first prepared address in that range, regardless of page-alignment, while the remaining entries will be page-aligned.

There are no explicit length fields in the mapping tables. Instead, entry lengths are implied by the entry's position in the table, the overall range length, and the page size. In the general case, the length of the first entry is to the next page alignment, the length of the intermediate entries (if any) is the page size, and the length of the last element is what remains by subtracting the previous lengths from the overall range length. If the prepared range fits within a single page, there is only one prepared entry and its length is equal to the preparedLength.

Specifying a physical mapping table implies that the transfer will bypass the main processor's MMU and data caches. Such transfers are performed with devices that fall into the DMA category of I/O devices. PrepareMemoryForIO and CheckpointIO take this into account when determining which cache operations are needed before and after the transfer.

- rangeInfo specifies the range or ranges to prepare. If the kIOMultipleRanges option is omitted, rangeInfo is interpreted as an AddressRange with the name “range.” If kIOMultipleRanges is specified, rangeInfo is interpreted as a MultipleAddressRange with the name “multipleRanges.” The firstPrepared specification determines the range table entry in which to begin preparation. Prior ranges, if any, will not be prepared and therefore need not have mapping table space allocated for them.

AddressRange fields have the following meanings.

- base specifies the lowest address in the range. If the kIOLogicalRanges option is omitted, base is treated as a physical address. If kIOLogicalRanges is specified, base is treated as a logical address in the address space specified by the addressSpace field.
- length specifies the length of the range.

A MultipleAddressRange specifies an array of AddressRanges, affecting a scatter-gather specification. MultipleAddressRange fields have the following meanings.

- entryCount specifies the number of entries in the rangeTable.
- rangeTable specifies the address of an array of AddressRange elements (an AddressRangeTable). See the AddressRange description, above. It is acceptable for the specified ranges to overlap either directly, or indirectly by being located on the same pages.

Finalizing I/O

```
typedef OptionBits IOCheckpointOptions;
enum {
    kNextIOIsInput      = 0x00000001,
    kNextIOIsOutput     = 0x00000002,
    kMoreIOTransfers    = 0x00000004
};
```

```
OSStatus CheckpointIO (IOPreparationID thePreparationID,
                      IOCheckpointOptions theOptions);
```

CheckpointIO performs the necessary follow-up operations for the specified device input/output transfer, and optionally prepares for a new transfer or reclaims the microkernel resources associated with the preparation. Call CheckpointIO to reclaim microkernel resources, even if the I/O is aborted.

Multiple concurrent preparations of memory ranges or portions of memory ranges are supported. In this case, cache actions are appropriate and individual pages are not unlocked until all transactions have been finalized.

thePreparationID is the IOPreparationID made for the input/output, as returned by a previous call to PrepareMemoryForIO. If the kMoreIOTransfers option is omitted, this ID becomes invalid after you call CheckpointIO.

theOptions specifies optional operations. Values for this field are defined by the IOCheckpointOptions type as follows:

- kNextIOsInput specifies that data will be moved into main memory.
- kNextIOsOutput specifies that data will be moved out of main memory.

Note that kNextIOsInput and kNextIOsOutput are completely independent. You may specify either, both, or neither. Specifying neither is useful for finalizing the previous transfer when the next transfer is not immediately pending.

- kMoreIOTransfers specifies that further I/O transfers will occur to or from the buffer. It is especially useful when the caller is unable to specify which direction the next transfer will be (i.e. neither kNextIOsInput nor kNextIOsOutput is specified), and is required if the next transfer direction is specified. If kMoreIOTransfers is omitted, all microkernel resources associated with preparation are reclaimed, including microkernel-allocated subsidiary structures and the IOPreparationID.

Memory Sharing

Memory sharing is a common requirement in device drivers, debuggers, and in client-server computing. The underlying method for sharing memory is to create areas that map the same backing store data into the various clients' address spaces. Because of the inherent memory caching, areas created this way use the same physical memory as well as the same backing store. Changes made to the memory in one address space are immediately present in the other address spaces. The microkernel provides various services to share memory using this method, each with its own merits and applications.

Global Areas

A "global area" is an area that appears in every address space, at the same location and with the same attributes, and is automatically added to new address spaces. Global areas are useful, for example, for mapping shared library code that needs to be equally available in all address spaces. They are made by specifying the kGlobalArea option (one of the AreaOptions) when creating the area.

Client-Server Areas

Certain servers benefit by providing their clients read-only access to the server's read-write data structures. This is simplified if the data appears at the same location in both the server's and the clients' address spaces, but with different memory access levels in each. The main hurdle is finding a location for the data that is available in the server and in all clients, present and future. "Memory reservations" address this problem. Reservations cordon off an address range such that areas will not be created there unless they are specifically placed there. For more information, see the section "Memory Reservations," later in this document.

Mapped Access to Other Address Spaces

It is sometimes useful to have on-going access to data in other address spaces. Although this can often be accomplished by creating an area with the same BackingObject and backing store offset as the area in the other space, this takes several microkernel calls and furthermore is impossible for areas without BackingObjects, such as resident areas. The memory system provides a routine so that cross-address space mapping can be established easily and for all types of areas.

```
OSStatus CreateAreaForRange
(
    KernelProcessID    owningKernelProcess,
    AddressSpaceID    otherSpace,
    LogicalAddress     otherBase,
    ByteCount          length,
    MemoryAccessLevel userAccessLevel,
    MemoryAccessLevel privilegedAccessLevel,
    ByteCount          guardLength,
    AreaOptions        options,
    LogicalAddress *   areaBase,
    AreaID             *   theArea);
```

CreateAreaForRange maps a logical address range from one space into another address space.

owningKernelProcess specifies the kernel process in whose address space to create the area.

otherSpace specifies the address space containing the range to map.

otherBase specifies the start of the range in otherSpace.

length specifies the number of bytes in the range. It must be non-zero.

otherBase and theLength will be page-aligned. This means that a bigger range than was specified may be mapped.

userAccessLevel and privilegedAccessLevel specify the kinds of memory references that non-privileged and privileged software are allowed to make in the area, respectively. References made in violation of the access level result in exceptions at the time of the access. For details, see the section “Memory Exceptions,” later in this document. If privilegedAccessLevel is more restrictive than userAccessLevel, privilegedAccessLevel will be made equal to userAccessLevel. Note that the underlying backing object may disallow certain access levels. CreateAreaForRange allows specification of two additional access levels: kInheritUserAccess and kInheritPrivilegedAccess. These specify that the access level is to be the same as the specified level in otherSpace. It’s possible to specify kInheritUserAccess as privilegedAccessLevel or vice versa. This is useful if the code calling CreateAreaForRange is running at a different privilege level than the code that was accessing the original area.

guardLength specifies the size, in bytes, of the excluded logical address ranges to place adjacent to each end of the area. The ranges, called *area guards*, are excluded to both privileged and non-privileged software. References to those addresses result in exceptions. For details, see the section “Memory Exceptions,” later in this document. guardLength will be page-aligned, if necessary. This means that the excluded ranges may be larger than is specified.

options specifies desired characteristics of the area being created. Values for this parameter are defined by the AreaOptions type. Note that some of these options will be inherited from the area containing the range being mapped, so they will be ignored by CreateAreaForRange.

- kZeroFill specifies that memory in this area should be initialized to zero. This option applies only to scratch areas (i.e. kNoBackingObjectID is specified in theBackingObject) and non-pageable areas (i.e. the kResidentArea option is specified). This option is inherited from the range being mapped.
- kResidentArea specifies that the data for this area must always be physically resident. These areas are never paged between memory and backing storage. This option is available only to privileged callers. This option is inherited from the range being mapped.
- kSparseArea specifies that the resources for the area be allocated on-demand. This option applies only to scratch areas (i.e. kNoBackingObjectID is specified in theBackingObject) and non-pageable areas (i.e. the kResidentArea option is specified). For scratch areas, sparseness means that the scratch backing object will be sparse, if possible. For resident areas, sparseness means that the physical memory

will be allocated by page faulting. This option is inherited from the range being mapped.

- `kPlacedArea` specifies that `areaBase` specifies where to create the area. The area will begin on the page specified by the `areaBase`. `CreateAreaForRange` fails and an error is returned if the area cannot be so positioned. The address corresponding to the beginning of the range will be returned in `areaBase`. Note that this will be exactly as specified only if `areaBase` and `otherBase` have the byte offset into their respective logical pages.

Note: Care should be taken when using the `kPlacedArea` option, as the specified location might be part of a memory reservation unknown to the caller. It is advisable to create a reservation for the range in which the area will be placed prior to creating the area. Reservations can be made for a specific address space or globally. For details, see the section “Creating Memory Reservations,” later in this document.

- `kGlobalArea` specifies that the data for this area is to be addressable from any address space. All address spaces will get access in accordance with the privileged and non-privileged access levels specified. The created area appears at `areaBase` in every address space.

`areaBase` is an output parameter indicating the address in the area corresponding to the beginning of the specified range. If the `kPlacedArea` option is specified, `areaBase` is also an input specifying where to position the area. See the description of `kPlacedArea` earlier in this section.

`theArea` is an output parameter indicating the area identifier that can be used for subsequent operations on the created area. A value of `kInvalidID` will be returned if `CreateAreaForRange` fails.

Copying Data Between Address Spaces

It is sometimes useful to read or write data in another address space. For example, a server may need to read or write client data, or a debugger might need to display or set data in the debugged address space. The memory system provides a routine to achieve this without the overhead of setting up a mapping and without the risk of encountering a memory access exception.

```
typedef OptionBits InterspaceCopyOptions;
enum
{
    kCheckSourceUserRights    = 0x00000001,
```

```
        kCheckDestinationUserRights    = 0x00000002
};
```

```
OSStatus InterspaceBlockCopy
    (AddressSpaceID      sourceAddressSpace,
     AddressSpaceID      targetAddressSpace,
     LogicalAddress       sourceBase,
     LogicalAddress       targetBase,
     ByteCount            length,
     InterspaceCopyOptions options);
```

InterspaceBlockCopy copies bytes from the specified source address space and range to the specified destination address space and range. Note that neither address space needs to be the current address space.

sourceAddressSpace specifies the address space containing the source range.

targetAddressSpace specifies the address space containing the destination range.

sourceBase specifies the start of the source range in theSourceAddressSpace.

targetBase specifies the start of the destination range in theTargetAddressSpace

length specifies the size, in bytes, of the range.

options specifies what access checks to apply to the copy operation. Values for this parameter are defined by the InterspaceCopyOptions type.

- kCheckSourceUserRights specifies that the user (non-privileged) access rights to the source address are to be checked. If not specified, the current execution mode's access rights are checked.
- kCheckDestinationUserRights specifies that the user (non-privileged) access rights to the destination address are to be checked. If not specified, the current execution mode's access rights are checked.

Memory Reservations

Certain servers benefit by providing their clients read-only access to the server's read-write data structures. This is simplified if the data appears at the same location in both the server's and the clients' address spaces, but with different memory access levels in each. The main hurdle is finding a location for the data that is available in the server and in all clients, present and future. Memory reservations address this problem. Reservations cordon off an address range such that areas will not be created there unless they are specifically placed there

(i.e. the area creator specifies the `kPlacedArea` `AreaOption`). Reservations can be made for a specific address space or globally.

Creating Memory Reservations

```
typedef OptionBits      ReservationOptions;
enum
{
    kPlacedReservation    = 0x00000001,
    kGlobalReservation     = 0x00000002,
    kGlobalAreaReservation = 0x00000004
};

OSStatus CreateAreaReservation(KernelProcessID    owningKernelProcess,
                              LogicalAddress *   base,
                              ByteCount          length,
                              ReservationOptions  options,
                              AreaReservationID * theReservation);
```

`CreateAreaReservation` reserves a logical address range such that no areas will be created within that range unless they are specified to be there. Areas are created at specific locations by using the `kPlacedArea` option, which is one of the `AreaOptions`.

`owningKernelProcess` specifies the kernel process in whose address space to reserve the range.

`base` is an output parameter indicating the beginning logical address of the reservation. If the `kPlacedReservation` option is specified, `base` is also an input specifying where to position the reservation. See the description of `kPlacedReservation` later in this section.

`length` is the number of bytes to reserve. It will be rounded up to a multiple of the logical page size. This means that the reservation may be larger than was specified.

`options` specifies desired characteristics of the reservation being created. Values for this parameter are defined by the `ReservationOptions` type.

- `kPlacedReservation` specifies that `base` specifies where to create the reservation. `base` and `length` will be page-aligned. This means that the reservation may be larger than was specified. `CreateAreaReservation` fails and an error is returned if the reservation cannot be so positioned. `base` will be set to the actual beginning of the reservation.

- `kGlobalReservation` specifies that the reservation is to apply across all existing and future address spaces. The reservation appears at base in every address space. Note that although the reservation is across all spaces, creating an area in it adds the area just to the address space specified by the area creator.

Global reservations, like global areas, are automatically added to new address spaces. This assures the server that the range will be available when a client in the new address space initializes its connection to the server.

- `kGlobalAreaReservation` specifies that the areas created in the reservation will be global, overriding the default behavior of global reservations. When this option is specified, the `kGlobalReservation` option also must be specified.

`theReservation` is an output parameter indicating the reservation identifier that can be used for subsequent operations on the reservation. A value of `kInvalidID` will be returned if `CreateAreaReservation` fails.

Deleting Memory Reservations

```
OSStatus DeleteAreaReservation (AreaReservationID theReservation);
```

`DeleteAreaReservation` destroys the specified memory reservation.

`theReservation` specifies the reservation to delete.

Obtaining Information About a Memory Reservation

```
struct ReservationInformation
{
    AddressSpaceID    addressSpace;
    LogicalAddress    base;
    ByteCount         length;
    ReservationOptions options;
};
typedef struct ReservationInformation ReservationInformation;

enum
{
    kReservationInformationVersion    = 1
};

OSStatus GetReservationInformation
        (AreaReservationID    theReservation,
         PBVersion            version,
         ReservationInformation * reservationInfo);
```

`GetReservationInformation` returns information about the specified memory reservation.

theReservation specifies the memory reservation for which to get the information.

version specifies the version number of ReservationInformation to be returned. This provides backwards compatibility. kReservationInformationVersion is the version of ReservationInformation defined in the current interface.

reservationInfo specifies where to return the information.

The fields of a ReservationInformation structure are:

- addressSpace - the address space in which the reservation exists.
- base - the logical address of the reservation.
- length - the size, in bytes, of the reservation.
- options - the options that were specified at the time the reservation was created.

Iterating Over All Memory Reservations Within an Address Space

```
OSStatus GetReservationsInAddressSpace
        (AddressSpaceID  addressSpace,
         ItemCount       requestedReservations,
         ItemCount *     totalReservations,
         AreaID *        theReservations);
```

GetReservationsInAddressSpace allows the caller to find the IDs of all reservations within a particular address space. For additional information about using iteration functions see the section "Some Basic Types," earlier in this document.

addressSpace indicates the address space of interest.

requestedReservations indicates the maximum number of reservation IDs that should be returned. This indicates the number of entries available at the location pointed to by theReservations.

totalReservations is filled in with the total number of reservations within the address space. If less than or equal to requestedReservations, all reservations were returned; if greater than requestedReservations, insufficient space was available to return all reservation IDs.

theReservations is filled in with the IDs of the reservations within theAddressSpace.

The status return is:

- noErr if addressSpace exists, whether or not all reservation IDs were returned.
- kernelIDerr if addressSpace doesn't exist

Memory Exceptions

The microkernel provides a mechanism to present exceptional hardware and software conditions to higher level software for resolution. (For information about this mechanism, see the section "Exceptions," earlier in this document.) The memory system employs this mechanism for address-space-related errors to be handled outside the memory system.

In particular, an address-space-related error results in an exception that can then be processed by an appropriate exception handler. The relevant ExceptionKinds are accessException, unmappedMemoryException, excludedMemoryException, readOnlyMemoryException, and unresolvablePageFaultException. The MemoryExceptionInformation structure defines additional information included in these exceptions. For information about how this structure is relayed to the handler, see the section "Exceptions," earlier in this document.

```
struct MemoryExceptionInformation
{
    AreaID          theArea;
    LogicalAddress  theAddress;
    OSStatus        theError;
    MemoryReferenceKind theReference;
};
typedef struct MemoryExceptionInformation MemoryExceptionInformation;

typedef unsigned long MemoryReferenceKind;
enum MemoryReferenceKind
{
    kWriteReference    = 0,
    kReadReference     = 1,
    kFetchReference    = 2
};
```

The fields of a MemoryExceptionInformation structure are:

- theArea - the area containing the logical address of the exception. This will be kInvalidID if the reference was made to an unmapped range of the address space.

Note: The value of this field is unpredictable if the memory access spanned area boundaries. The use of area guards reduces the probability of such accesses.

- theAddress - the logical address of the exception.
- theError - the status for unresolvablePageFault.
- theReference - the type of memory reference that resulted in the exception.

The address-space-related values of the ExceptionKind type are:

- accessException - the reference resulted in a page fault because the physical address was not accessible (i.e., it was a “hard fault”).
- unmappedMemoryException - the reference was to an address which is not part of any area in the address space.
- excludedMemoryException - the reference was to an area whose access level prevents any access (the ExcludedMemory access level), or to a area guard.
- readOnlyMemoryException - the reference was to an area whose access level prevents write accesses (the ReadOnlyMemory access level).
- unresolvablePageFaultException - the reference resulted in a page fault that could not be resolved. theError field in the MemoryExceptionInformation indicates why the fault was not resolved.

The values of the MemoryReferenceKind type are:

- kWriteReference - the reference was an attempt to modify data.
- kReadReference - the reference was an attempt to acquire data.
- kFetchReference - the reference was an attempt to acquire a processor instruction.

Note: The ability to distinguish instruction fetches from read references is processor dependent. Consequently, some implementations may report an instruction fetches as a read reference.

MESSAGING

For the purposes of discussion, the message system is divided into sections on the management of message ports, objects, and the messaging operations.

Message Port Management

Message ports are abstract entities used to receive messages. The microkernel provides operations for the creation, deletion, and maintenance of message ports.

Ports, like other kernel objects are referenced by ID.

Creating Message Ports

When a message port is created, it contains all microkernel resources needed to receive and reply to messages that are sent synchronously. Additional resources are needed by the microkernel for each asynchronous operation (sends and receives) that occur simultaneously.

```
OSStatus CreatePort (PortOptions      options,  
                    PortID *        thePort);
```

options control the details of port creation. Currently no options are supported and a value of kNilOptions should be specified.

thePort is updated with the ID of the newly created message port.

Deleting Message Ports

Deletion of a message port deletes all associated message objects. As a result of deleting the associated objects, any outstanding send requests to those objects, and therefore the port, are completed with appropriate status. Further, any outstanding receive requests to the port are similarly completed with appropriate status.

After deletion, the port's ID becomes invalid and subsequent attempts to use it result in an error.

```
OSStatus DeletePort (PortID thePort);
```

Obtaining Information About a Port

You can request information about a given port. Information regarding the current state of the message port and how it was created are returned.

```
typedef struct MessagePortInformation
{
    KernelProcessID    owningKernelProcess;
    ItemCount          objectCount;
    ItemCount          pendingReceives;
    ItemCount          pendingSends;
    ItemCount          pendingReplies;
    ItemCount          transactionCount;
    ItemCount          blockedAsyncSenders;
    ItemCount          blockedAsyncReceivers;
} MessagePortInformation;

enum
{
    kPortInformationVersion = 1
};

OSStatus GetPortInformation (PortID          thePort,
                             PBVersion      version,
                             MessagePortInformation * portInfo);
```

thePort is the ID of a message port about which you want information.

version specifies the version number of PortInformation to be returned. This provides backward compatibility. kPortInformationVersion is the version of PortInformation defined in the current interface.

portInfo is the address of a MessagePortInformation record that will be filled in with information about the message port.

After a call to GetPortInformation, the portInfo parameter is filled in with the following information:

- owningKernelProcess is the ID of the kernel process that created the message port.
- objectsCount is the number of message objects that are currently associated with the message port.
- pendingReceives indicates the number of receive requests that have been made of the port but have not yet been matched with any message.
- pendingSends indicates the number of send requests that have been made to message objects associated with the port but have not yet been matched to any receive request.

- pendingReplies indicates the number of send requests that have been made to message objects associated with the port and have been received but to which no reply has been issued.
- transactionCount is the total number of send-receive-reply transactions that have taken place across this message port since it was created.
- blockedAsyncSenders indicates the number of asynchronous senders that have issued requests but been blocked waiting for message system resources.
- blockedAsyncReceivers indicates the number of asynchronous receivers that have issues requests but been blocked waiting for message system resources.

Iterating Over Message Ports

You can find all the message ports in the system by using the following function.

```
OSStatus GetPortsInSystem (ItemCount      requestedPorts,
                          ItemCount *    totalPorts,
                          PortID *      thePorts);
```

GetPortsInSystem returns the IDs of all the ports in the system.

requestedPorts indicates the maximum number of port IDs that should be returned. This indicates the number of entries available at the location pointed to by thePorts.

totalPorts is filled in with the total number of ports in the system. If less than or equal to requestedPorts, all port IDs were returned; if greater than requestedPorts, insufficient space was available to return all port IDs.

thePorts is filled in with the IDs of the ports the system.

Message Object Management

Message objects are the abstract entities to which messages are sent. Objects are associated with exactly one port. This association may be changed. Messages sent to objects are received from the object's associated port.

Message objects contain a reference constant. This reference constant, typically a control block address, is copied from the object into the message at the time a message is sent through the object to a port.

Message objects may have a designated client kernel process. Newly created message objects have no such client designated. Part of the processing performed by the microkernel during kernel process termination includes sending messages to any objects whose client is the terminating kernel process. This ability allows servers to reclaim message objects whose clients have terminated.

The microkernel provides services for the creation, deletion, and maintenance of message objects.

Like all kernel objects, message objects are referenced by ID.

Creating Message Objects

Creation of message objects requires that you specify a port with which the object is initially associated. You must also specify an initial value for the object's refcon. Once created, the message object is immediately eligible to be the target of send requests.

```
OSStatus CreateObject (PortID      port,
                      ObjectRefcon refcon,
                      ObjectOptions options,
                      ObjectID *    theObject);
```

port indicates the port with which the message object is to be associated. Messages sent to the object being created will appear at this port.

refcon indicates the value of the message object's refcon. This value will be copied from the object being created into messages at the time they are sent through the object and placed into the object's port.

options is currently unused.

theObject is updated with the ID of the newly created message object.

Deleting Message Objects

Deletion of a message object implies replying to any messages that have been sent to the object but have not as yet been received from the object's port. These send requests are made to complete with appropriate status. After deletion, the object's ID becomes invalid and subsequent attempts to use it are erroneous. Such attempts usually result in errors. Deletion of a message object also unlocks the object. Any tasks waiting for the lock are given an error result.

```
OSStatus DeleteObject (ObjectID theObject);
```

Locking Message Objects

Message objects can be locked. Once locked, messages sent to the object cannot be received until the object is unlocked. Multiple tasks may attempt to lock an object. However, only one task is granted the lock. Any other tasks are blocked in priority order awaiting the lock.

```
typedef OptionBits ObjectLockOptions;
enum
{
    kLockObjectWithOneMessage = 0x00000001
};
```

```
OSStatus LockObject (ObjectID theObject,
                    ObjectLockOptions options,
                    Duration timeout);
```

theObject is the ID of the message object to be locked.

options controls the behavior of the request to lock the designated object. The kLockObjectWithOneMessage option controls the number of messages that have been received but not replied when the lock request is satisfied. If this option is specified, the number of such messages is exactly one; in the absence of this option, the number of such messages is exactly zero.

timeout places a maximum waiting limit on the LockObject operation. If the value of timeout is exceeded, LockObject fails and returns an error.

Unlocking Message Objects

The UnLockObject service is used to release the lock on a message object.

```
OSStatus UnLockObject (ObjectID theObject);
```

theObject specifies the locked object that is to be unlocked.

Obtaining Information About an Object

Given the ID of a message object, you can obtain the ID of the port with which it is currently associated, the kernel process that is the object's client, and the refcon currently associated with the object.

```
OSStatus GetObjectInformation (ObjectID      theObject,
                              PortID *     port,
                              ObjectRefcon * refcon);
```

theObject is the ID of an object about which information is to be returned.

port is update to indicate the ID of the Port to which this object belongs.

refcon is updated to indicate the object's current refcon.

Changing Information About an Object

```
enum
{
    kSetObjectPort      = 0x00000002,
    kSetObjectRefcon    = 0x00000004
};
typedef OptionBits    SetObjectOptions;

OSStatus SetObjectInformation (ObjectID      theObject,
                              SetObjectOptions options,
                              PortID      port,
                              ObjectRefcon refcon);
```

theObject is the ID of an object about which information is to be returned.

options controls which, if any, of the object's information is changed. This value is a mask formed by ORing together the values kSetObjectPort and kSetObjectRefcon.

port is the ID of a port to which this object will be moved. This value is only used if options includes the kSetObjectPort bit. Changing an object's port causes any unreceived messages to be forwarded from the old to new port.

refcon is the refcon value that will be associated with the object. This value is only used if options includes the kSetObjectRefcon bit.

Iterating Over Objects

You can iterate over all of the message objects associated with a particular message port.

```
OSStatus GetObjectsInPort (PortID          thePort,
                          ItemCount       requestedObjects,
                          ItemCount *     totalObjects,
                          ObjectID *      theObjects);
```

GetObjectsInPort allows the caller to find the IDs of all tasks within a particular port. For additional information about using iteration functions see the section “Some Basic Types,” earlier in this document.

thePort indicates the port of interest.

requestedObjects indicates the maximum number of object IDs that should be returned. This indicates the number of entries available at the location pointed to by theObjects.

totalObjects is filled in with the total number of objects on the port. If less than or equal to requestedObjects, all objects were returned; if greater than requestedObjects, insufficient space was available to return all object IDs.

theObjects is filled in with the IDs of the objects on the port.

The status return is:

- noErr if thePort exists, whether or not all object IDs were returned.
- kernelIDerr if thePort doesn't exist

About Message Transactions

A message transaction is begun with a send. Once begun, the transaction is *in-progress* until it completes. Transactions are completed by either a reply or by cancellation of the send request.

When the receiver replies to a received message, the reply includes a status indication. This status indication is called the *reply status*.

Message IDs

All message transactions can be identified by a particular message ID. The ID of an individual message transaction is used to query or alter the state of a transaction. With the exception of the synchronous send operation, every message system operation either requires that you specify a message ID or returns a message ID.

Message Types

Each message that is sent is accompanied by a message type. When a server makes receive requests, it may indicate that it only wants to receive messages of a certain type. These message types help to classify the message in a manner agreed upon between the client and server. You can use message types to prioritize message importance, to differentiate between kinds of requests, or for other purposes.

A message type is a 32-bit value that is interpreted as an array of 32 bits. A sender specifies the type of message being sent by passing a message type with one or more bits set. A receiver specifies the type of message it wishes to receive by specifying a message type with one or more bits set. A particular receive request will only be satisfied if the logical AND of the sender's message type and the receiver's message type is non-zero.

```
typedef UInt32    MessageType;
```

Notice that a message sent with a message type value of zero cannot match any receiver using the rules described in this section. However, a receive request that specifies a message type value of 0xFFFFFFFF will match any message, including receive requests that have a message type of zero.

Kernel Messages

The microkernel reserves the most significant bit of the message type parameter to indicate that the message is a *kernel message*. All messages sent by the microkernel are of the kernel message type. The next three most significant bits are reserved for future use. These messages are used to perform various system management functions such as canceling requests. The use of these messages is discussed in the section "Canceling Message Requests," later in this document.

Note: Clients of the microkernel should not send messages that have the kernel message type.

```
enum
{
    kernelMessageType          = 0x80000000,
    kAllNonKernelMessageTypes = 0xFFFFFFFF,
    allMessages                 = 0xFFFFFFFF
};
```

Each kernel message begins with a common header that allows the various messages to be distinguished. Individual kernel messages are described throughout this document. Some definitions useful when handling kernel messages are:

```
typedef struct KernelMessageHeader
{
    UInt32      messageCode;
} KernelMessageHeader;
```

Sending Messages

Clients request actions by sending messages to objects. For example, if you want to read ten bytes of data from serial port A, you send a message to the object that represents serial port A. The message describes the nature of the actions you want the object to perform. In this case, the message would indicate a read request with a byte count of ten.

From the perspective of the microkernel, messages are simply a set of memory locations described by a single address/byte count pair. It is the responsibility of the sender to insure that, from the time the send is initiated until the time the send completes, the contents of those memory locations remain intact. This means, for example, that a message that is sent asynchronously should not be allocated on the stack of the sender unless the sender can guarantee that the contents of the stack frame will remain valid until a reply is received or the send is canceled.

The descriptive nature of messages form an agreement between client and server; they are not examined or interpreted by the microkernel.

Send Options

When you send a message, you can control certain aspects of the message transmission through use of the SendOptions parameter. These options are described below:

```
typedef OptionBits SendOptions;
```

```

enum
{
    kSendTransferKindMask      = 0x00000003,
    kSendByChoice              = 0x00000000,
    kSendByReference           = 0x00000001,
    kSendByValue               = 0x00000002,
    kSendIsBuffered            = 0x00000003,

    kSendIsPrivileged          = 0x00000008,
    kSendIsAtomic              = 0x00000010,
    kSendPtrsAddressable       = 0x00000040,
    kSendPtrsNeedAccessCheck   = 0x00000080
};

```

- `kSendTransferKindMask` identifies the subfield within the `SendOptions` that specifies how the data is to be transferred between sender and receiver.
- `kSendByChoice` causes the microkernel to choose `kSendByReference` or `kSendByValue`, whichever is faster.
- `kSendByReference` causes just the address of the sender's message to be placed into the receiver's buffer. If the message sender is in an address space different from that of the message receiver, this option causes the message to be mapped into the receiver's address space. Such mappings are eliminated when the transaction, initiated by the send. This option must be specified if the intent of the transaction is for the sender to receive data in the message buffer placed there by the recipient.
- `kSendByValue` causes the microkernel to copy the contents of the sender's buffer into the receiver's buffer.
- `kSendIsBuffered` option causes the microkernel to buffer the message internally so that the sender can reuse the buffer contents immediately. This is only useful for asynchronous sends. There's no guarantee that the microkernel can allocate the required memory, and it may enforce a maximum size on the message. If the message exceeds the maximum size the microkernel is willing to buffer, or exceeds the memory the microkernel can allocate, the caller will be blocked and the message delivered synchronously.
- The `kSendIsPrivileged` option bit is set by the microkernel on behalf of a privileged sender task. If a non-privileged sender attempts to set this option bit, the microkernel will clear it before passing the message to a receiver.
- The `kSendIsAtomic` option bit causes the microkernel to lock the object until a Reply is issued.
- `kSendPtrsAddressable` is set by the microkernel if the sender's address space is directly addressable by the receiver. This is true when both sender and receiver are user mode tasks in the same address space, or when both are privileged tasks, or when the receiver is an accept function.

- `kSendPtrsNeedAccessCheck` is set by the microkernel if attempts by the receiver to access data in the sender's address space must first have access checks applied explicitly by the receiver. `kSendPtrsNeedAccessChecks` is false whenever `kSendPtrsAddressable` is true *except* that `kSendPtrsNeedAccessChecks` is true if the sender is a user mode task and the receiver is an accept function. In this case, the sender's address space is accessible to the accept function, but because the accept function is privileged and the sender is not, their access rights to a particular address may not be the same.

Synchronous Sends

Synchronous message sends behave like a subroutine call. An optional time-out value may be used by the sender to place an upper limit on the overall transaction. Synchronous sends cause the sending task context to block until the receiver has issued a reply or the request has timed out.

Should the time limit be exceeded, the message system will cancel the incomplete message transaction. If the message has been received, the effect of the cancellation is up to the receiver. Cancellation is described in the section "Canceling Message Requests." later in this document.

Synchronously sent messages are placed at the end of the message queue of the port associated with the object to which the message is sent. The message will be processed when it is matched to a receiver. This matching is controlled by message type and order within the queue.

Synchronous send requests cannot be explicitly canceled. They are only canceled implicitly as a result of a timeout.

The microkernel may decide to map the sender's reply and contents buffers into the receiver's address space. Any such mapping is eliminated upon reply.

```
OSStatus SendMessageSync      (ObjectID      object,
                               MessageType    theType,
                               LogicalAddress messageContents,
                               ByteCount     messageContentsSize,
                               LogicalAddress replyBuffer,
                               ByteCount     * replyBufferSize,
                               SendOptions   options,
                               Duration       timeout);
```

`object` specifies the destination object.

`theType` specifies the type of message.

messageContents specifies the address of the outgoing message data. A null value indicates no contents. The sender should not access this buffer until the transaction completes.

messageContentsSize specifies the length of the outgoing message data.

replyBuffer specifies the address of a buffer to be used for the server's reply data. A null value indicates no reply data is desired. The microkernel may choose to map this buffer into the receiver's address space. The sender should not access this buffer until the transaction completes.

replyBufferSize specifies the size of the reply buffer. This parameter is both an input and output value. On input, it specifies the size of the sender's reply buffer. Upon completion of the send, it holds the number of bytes transferred into the reply buffer.

options specifies a bit mask of send options. These options are passed along to the server at the time it receives the message.

timeout specifies a time after which an automatic cancellation is performed by the message system. A value of durationForever specifies no such automatic cancellation. A value of durationImmediate specifies that a cancellation take place if the message cannot be immediately matched to a receiver. If such a match is possible, no further time constraint is placed upon the transaction. For a complete description of the type Duration, see the section "Some Basic Types," earlier in this document.

Asynchronous Sends

An asynchronous send allows the sending task context to continue execution while the transaction remains incomplete. You'll receive notification that the transaction has completed in a manner governed by the kernel notification you provide at the time you send the message.

Asynchronously sent messages are placed in the message queue of the port associated with the object to which the message is sent. The message will be processed when it is matched to a receiver. This matching is controlled by message type and order within the queue.

The microkernel may decide to map the sender's reply buffer into the receiver's address space. Any such mapping is eliminated upon reply.

```
OSStatus SendMessageAsync (ObjectID          object,
                           MessageType       theType,
                           LogicalAddress    messageContents,
                           ByteCount        messageContentsSize,
                           LogicalAddress    replyBuffer,
                           ByteCount        replyBufferSize,
                           SendOptions       options,
                           const KernelNotification * notification,
                           ByteCount *      replySize,
                           MessageID *     heMessageID);
```

object specifies the destination object.

theType specifies the type of message.

messageContents specifies the address of the message data.

messageContentsSize specifies the length of the message data.

replyBuffer specifies the address of a buffer to be used for the server's reply data. A null value indicates no reply data is desired. The microkernel may choose to map this buffer into the receiver's address space. The sender should not access this buffer until the transaction completes.

replyBufferSize specifies the size of the reply buffer.

options specifies a bit mask of send options. These options are passed along to the server at the time it receives the message.

notification specifies an asynchronous event completion record. This event will be delivered when the transaction completes.

replySize specifies the address where the actual size of the reply will be stored. Upon completion of the send, it holds the number of bytes transferred into the reply buffer.

theMessageID specifies the address of a message id. The message system stores an ID for the transaction at this address. This ID can be used by the sender to cancel the transaction.

Receiving Messages

Servers must inform the message system that they want to receive messages. This is done in one of three ways: synchronous receives, asynchronous receives, or message acceptance functions.

Synchronous and asynchronous receives are requests for a single message of one or more message types. The receive request is satisfied if a message of suitable type is already present in the queue at the time the receive request is made or arrives in the queue within the time limit, if any. Once the match between sent message and receive request has been made, processing of the message happens in the context of the task that made the receive request. To receive subsequent messages the receiver must make additional receive requests.

Multiple receives, either synchronous or asynchronous, may be pending upon a single port simultaneously. These requests can be for different message types or for the same message type. When new messages arrive at the port, receivers are matched in the order that their receive requests were made.

Message acceptance is quite different from synchronous and asynchronous receive requests. When you register a message acceptance function with a message port, that function will be called for every message sent to that port as long as the message type specified by the sender matches the message type specified by the receiver. When you no longer want to accept messages from the port, you must unregister your acceptance function.

Note: Unlike synchronous and asynchronous receive requests, an acceptance function does not receive messages that were in the port's message queue prior to the time the function was registered. For this reason, it is strongly suggested that the server register any acceptance function just after creating the port and prior to creating

any message objects. This will ensure that no messages are queued before the acceptance function is registered.

Acceptance functions are always called in the task context of the sending task. The function executes in supervisor mode on the microkernel stack of the current task. For that reason, the acceptance function and everything it calls must reside in globally shared memory so that it can be addressed successfully. Because an acceptance function runs in its client's task context, its client's address space is directly addressable. An acceptance function is the only privileged software that can directly dereference pointers passed to it by a non-privileged client.

Receive Options

When you receive a message, you can control certain aspects of the message transmission through use of the `ReceiveOptions` parameter. These options are described below:

```
typedef OptionBits ReceiveOptions;
enum
{
    kReceiveNoAddressTranslation = 0x00000002,
};
```

- By default, the message system makes the sender's message addressable by the receiver. This involves mapping the message contents if the sender and receiver share the same address space or copying the message contents if the sender and receiver reside in different address spaces. To prevent either of these operations, the receiver can specify the `kReceiveNoAddressTranslation` option. In this case, the receiver must insure that the contents are addressable before it accesses the message.

Message Control Blocks

When you receive a message, the message system provides you with a control block that describes the received message. This control block indicates both the address and length of the sender's message and provides additional information including the message type and options specified by the sender.

The control block is built in different places depending on the kind of receive operation you make. Synchronous and asynchronous receives cause the control block to be constructed in a buffer you must supply when you make the receive request. Acceptance function receives cause the control block to be constructed within the microkernel prior to calling your function.

```
typedef struct MessageControlBlock
```

```

{
  MessageID          message;
  AddressSpaceID    addressSpace;
  AddressSpaceID    sendingKernelProcess;
  ObjectRefcon      refcon;
  SendOptions       options;
  MessageType       theType;
  ByteCount         messageContentsSize;
  LogicalAddress    messageContents;
  ByteCount         replyBufferSize;
  LogicalAddress    replyBuffer;
  OSStatus          currentResults;
  SInt32            reserved;
} MessageControlBlock;

```

- `message` is an ID that represents this send-receive-reply transaction. As the receiver of a message, you use this ID to reply to the message. If the message was sent asynchronously, this is the same ID returned to the sender at the time of the send. The sender can use the ID to cancel the send. As the receiver of a message you should be prepared to handle cancel requests for messages that you have received but to which you have not yet replied; you'll need this ID to process such cancel requests. For information about canceling a message, see the section "Canceling Message Requests," later in this document.
- `addressSpace` is the ID of the sending task's address space. This field is provided so that servers can map portions of the sender's address space.
- `sendingKernelProcess` is the ID of the sending task's kernel process. This field is provided so that servers can verify object permissions and clients.
- `refcon` is the refcon of the object to which the sender sent the message.
- `options` are the send options as specified by the sender.
- `theType` is the message type as specified by the sender.
- `messageContentsSize` is the message size, in bytes, as specified by the sender.
- `messageContents` is the address at which you can find the message. This field points to the sender's original message or to the buffer you specified at the time you made the receive request.
- `replyBufferSize` is the size of the sender's reply buffer specified in the `SendMessage` request. A zero value indicates that the sender does not have a reply buffer.
- `replyBuffer` is the address at which you can find the sender's reply buffer. This field points to the sender's original reply buffer or is nil. A nil value indicates that the sender either doesn't have a reply buffer or that the microkernel chose not to map it. If the reply buffer exists (that is, if `replyBufferSize` is not zero), the server can use the `ReplyToMessage` service to copy the reply data. For more information about this service, see the section "Replying to Messages," later in this document.

- `currentResults` is the status supplied by a `ReplyToMessage`.

Receiving Messages Synchronously

Synchronous receives cause the receiving task to block until a message arrives at the specified port that can be matched to the receive request. An optional timeout value may be used to place an upper limit upon the time that the receive waits for incoming messages. Should this time limit be exceeded, the message system removes the request and unblocks the calling task. In this case, you'll receive an indication that no messages of suitable type arrived and that the request was terminated.

```
OSStatus ReceiveMessageSync (PortID          port,
                             MessageType     theTypes,
                             MessageControlBlock * controlBlock,
                             LogicalAddress  buffer,
                             ByteCount      bufferSize,
                             ReceiveOptions  options,
                             Duration        timeout);
```

`port` specifies the port from which you wish to receive messages.

`theTypes` specifies the type of message you wish to receive.

`controlBlock` specifies the address of the message control block. The control block describes the message you've received.

`buffer` specifies the address of a receive buffer. If the message sender did not specify the `sendByReference` option, the contents of the sender's message may be copied into this buffer. You should always check the message control block to see where the data has actually been received to.

`bufferSize` specifies the total size of the receive buffer. When the receive completes, the actual number of bytes received is placed in the message header.

`options` specifies a bit mask of available receive options.

`timeout` specifies a time after which an automatic cancellation is performed by the message system. A value of `durationForever` specifies that no such automatic cancellation should take place. A value of `durationImmediate` specifies that a cancellation should take place if the receive request cannot be immediately matched with a message already at the port. For a complete description of the type `Duration`, see the section "Some Basic Types," earlier in this document.

Receiving Messages Asynchronously

An asynchronous receive allows the receiving task context to continue execution while awaiting the arrival of a message. You'll receive notification of a suitable message in a manner governed by the kernel notification you provide at the time you make the request.

When you make asynchronous receive requests, you'll receive an ID that may be used at a later time to cancel the request. This ID remains valid until you receive a message or until you cancel the receive request.

```
OSStatus ReceiveMessageAsync (PortID          port,
                             MessageType      theType,
                             MessageControlBlock * controlBlock,
                             LogicalAddress   buffer,
                             ByteCount       bufferSize,
                             ReceiveOptions   options,
                             const KernelNotification * notification,
                             ReceiveID *     theReceive);
```

port specifies the port from which you wish to receive messages.

theType specifies the type of message to you wish to receive.

controlBlock specifies the address of the message control block. The control block describes the message you've received.

buffer specifies the address of a receive buffer. If the message sender did not specify the `sendByReference` option, the contents of the sender's message may be copied into this buffer. You should always check the message control block to see where the data has actually been received to.

bufferSize specifies the total size of the receive buffer. When the receive completes, the actual number of bytes received is placed in the message header. Note that the `bufferSize` must be at least the size of a message control block and must be larger than a message control block to actually receive any of the message from the sender.

options specifies a bit mask of available receive options.

notification specifies a kernel notification that will be delivered when the receive request completes.

theReceive specifies the address of receive ID. The message system stores the ID of the in-progress receive at this address. The receiver can use this ID to cancel the in-progress receive operation.

Accepting Messages

Accepting messages establishes an acceptance function as the recipient of all messages of a given type that are sent to a specific message port. At the time a message arrives at a port, the port is examined for eligible receivers. If an acceptance function has been registered and matches the type of the sent message, it is called in the context of the sending task, in lieu of any of other receivers that may be present. At most, one acceptance function can be registered with a given message port.

When the message acceptance function is called, it is provided with two parameters. The first is a pointer to the message control block that describes the message. The second parameter is a refcon defined when the acceptance function is installed. The function is called in supervisor mode and runs on the microkernel mode stack of the sending task. Therefore, such routines must be loaded into globally shared memory. This must be done using Server Manager or Code Fragment Manager services .

Just like other message receivers, the message acceptor must reply to the sent message. Acceptance functions can use the ReplyToMessage microkernel service to explicitly reply to the message. (For information about this service, see the section "Replying to Messages," later in this document.)

Acceptance functions can cause an implicit reply to the message being processed by returning any OSStatus value other than kernelIncompleteErr. The status returned will be used by the microkernel as if it were passed in an explicit call to ReplyToMessage.

If, at the time the acceptance function returns, no explicit reply has been generated and the status value returned is kernelIncompleteErr, the sending task will be blocked if the send operation was synchronous. Under these conditions, the sending task will not become eligible for execution until either a reply is issued or the time limit specified by the sender is exhausted.

While an acceptance function is executing, the sending task is still preemptable. This means that a separate task could perform another send operation, which causes the acceptance function to be re-entered. Because of this, your acceptance function, and all other software it calls, must be reentrant.

When implementing lightweight services that must be serialized, it is frequently the case that upon accepting a message the acceptance function transfers control to secondary interrupt level to serialize requests. As a further optimization the message system allows you to specify that your acceptance function should be

called at secondary interrupt level. This option is specified when you register your acceptance function with the message port.

Whenever an acceptance function is registered for a particular port, an exception handler must also be registered. This exception handler will be invoked should an exception arise during the processing performed by the acceptance function. This handler receives control in lieu of the sending task's handler.

```
typedef OSStatus (*MessageAcceptProc)
                                     (MessageControlBlock * message,
                                      void *                refcon);

typedef OptionBits    AcceptOptions;

enum
{
    kAcceptFunctionIsResident    = 0x00004000
};

OSStatus AcceptMessage (PortID        port,
                       MessageType    theType,
                       MessageAcceptorProc proc,
                       ExceptionHandler handler,
                       AcceptOptions  options,
                       void *        acceptRefcon);
```

port specifies a port from which messages are to be accepted.

theType specifies a bit mask of acceptable message types.

proc specifies an acceptance function.

handler specifies an exception handling routine that will receive control should the acceptance function cause an exception.

options specifies a bit mask of available accept options.

- kAcceptFunctionIsResident causes the microkernel to keep the acceptance function's stack resident.

acceptRefcon specifies the refcon that will be passed to the acceptance function when it's called.

Replying to Messages

After a message has been sent and received, the receiver performs the request implied by the message. When the request has been processed, or if the receiver decides it cannot complete the request, the receiver must inform the sender of the transaction's status. This is done by replying to the message. Replying to a message completes a message transaction.

You must identify the message to which you are replying; this is done by passing the message ID. The message ID is provided to a receiver in the message control block.

In addition to the message ID, the receiver must supply a status value. This status is not examined or interpreted by the message system. Rather, it is passed back to the sender.

As a result of a reply, the message system takes several actions. If the send was performed synchronously, the status value is stored in the requested location and the sending task is unblocked. If the send was performed asynchronously, the requested notification is delivered. Finally, any message buffer mapping that was created is eliminated.

If a reply buffer was specified by the sender, the microkernel may choose to supply its address and size in the MessageControlBlock. If the sender's reply buffer address is supplied by the microkernel, the receiver may use it to directly transfer the result data. If the sender's reply buffer address is not supplied by the microkernel, the receiver can use the reply to copy the reply data back to the sender.

```
OSStatus ReplyToMessage      (MessageID      theMessage,
                             OSStatus      status,
                             LogicalAddress replyBuffer,
                             ByteCount     replyBufferSize);
```

theMessage specifies the ID of a message.

status specifies the status value to return to the sender. If the message was sent synchronously, status becomes the return value to SendMessageSync. If the message was sent asynchronously, status is returned through the kernel notification record's status field.

replyBuffer specifies the address of the reply data. The microkernel will copy the data from the receiver to the sender. A nil value indicates no reply data should be delivered to the sender.

replyBufferSize indicates the size of the reply data. If replyBufferSize is greater than the size of the sender's reply buffer, the reply data is truncated to the sender's size.

Replying to a Message And Receiving Another Message

The message system also provides a special purpose service that combines a reply with a synchronous receive. The reply is performed first, followed by the synchronous receive. All parameters behave as documented for the ReplyToMessage and ReceiveMessage services, which are described in the sections "Replying to Messages" and "Receiving Messages Synchronously," earlier in this document. A MessageID of nil tells the microkernel to skip the reply step.

```
OSStatus ReplyToMessageAndReceive
    (MessageID          theMessage,
     OSStatus          results,
     LogicalAddress     replyBuffer,
     ByteCount         replyBufferSize,
     PortID            port,
     MessageType       theType,
     MessageControlBlock * controlBlock,
     LogicalAddress     buffer,
     ByteCount         bufferSize,
     ReceiveOptions     options,
     Duration           timeout);
```

Obtaining Information About a Message

Given the ID of a message, you can get information about it. The ID of the kernel process that sent the message is included in the MessageControlBlock, but the sending task and the object to which the message was sent can only be obtained by calling GetMessageInformation.

```
struct MessageInformation
{
    ObjectID          object;
    TaskID           sendingTask;
    KernelProcessID  sendingKernelProcess;
};
typedef struct MessageInformation MessageInformation,
    *MessageInformationPtr;

OSStatus GetMessageInformation (MessageID          theMessage,
                               PBVersion         version,
                               MessageInformation * messageInfo);
```

theMessage is the ID of the message about which you want information.

version specifies the version number of MessageInformation to be returned. This provides backward compatibility. kMessageInfoVersion is the version of MessageInformation defined in the current interface.

After a call to GetMessageInformation, the messageInfo field is filled in with the following information:

- object is the ID of the object to which the message was sent.
- sendingTask is the ID of the task that sent the message
- sendingKernelProcess is the ID of the kernel process that sent the message. This is the same as the sendingKernelProcess field in the MessageControlBlock.

Canceling Message Requests

At times, it is necessary to withdraw pending asynchronous send or receive requests. Message requests might be withdrawn, for example, when a particular service shuts down its operation. Withdrawing these requests is called *canceling* them.

Because you may have several pending requests simultaneously, you must indicate the particular request you wish to cancel. The ID returned at the time the request was made is used to cancel that particular request.

Cancellation of a send or a receive request causes the request to complete with an error indication. The error indication is supplied along with the ID. The microkernel does not interpret the error indication.

Cancellation of any request causes an implicit race condition between the client and the server. It is possible that the server completes the request at the same time as the client attempts to cancel the request. It is also possible for servers to ignore cancellation requests and finish the original request. In either case, it is the responsibility of the client to correctly handle these race conditions and understand that cancellation may result in either normal or abnormal completion of the request being canceled.

Send and receive requests are canceled by the microkernel as a side effect of certain operations. These operations are:

- Deletion of a message port causes all receive requests of that port to be canceled.
- Deletion of a message port causes all messages sent to objects associated with the port to be canceled.

- Deletion of a message object causes all unreceived messages sent to that object to be canceled.
- Termination of a task causes all messages sent by that task, that have not been replied to, to be canceled.
- Termination of a task causes all receive requests made by that task to be canceled.
- Timeout of a message send request causes that request to be canceled.
- Timeout of a message receive request causes that request to be canceled.
- Explicit cancellation requests cause the associated request to be canceled.

Send Message Cancellation

Cancellation of an asynchronous send message request occurs in one of two ways. If the message has not yet been matched to a receiver, the message is removed from the port and no special actions are taken. The sender is notified that the send operation has completed with status that indicates the send was canceled.

If the message has been matched with a receiver but has not as yet been replied to, the cancellation process is quite different. In this case, a kernel message containing the ID of the message being canceled is placed in the message queue of the port from which the message was received. It is up to the receiver to process the kernel message appropriately. If the receiver has replied to the message being canceled prior to receiving the kernel cancellation message, the microkernel removes the cancel message from the queue so that it is never seen by the receiver.

If, however, the cancellation message is received prior to the time the receiver replies to the message, the receiver should make every attempt to abort whatever work is in progress on that message. Once processing has stopped for the message being canceled, the receiver must issue replies for both the canceled message and the kernel cancellation message.

The status given when replying to a canceled message should convey that the request was canceled. The status given when replying to the kernel cancellation message should indicate that the cancellation was handled successfully. If desired, you can reply to only the kernel cancellation message with the status value `kernelCanceledErr`. This value causes the message system to reply to the canceled message with the `kernelCanceledErr` status value, saving you the effort of replying to both messages.

The microkernel service that cancels an asynchronous message send operation is itself synchronous. It does not return to the caller until that send operation has completed.

```
OSStatus CancelAsyncSend (MessageID theMessage,
                          OSStatus  status);
```

theMessage is the ID of the asynchronous send request that is to be canceled.

status is the status value with which the send is to complete. This is the value that the asynchronous notification, if any, will deliver.

Receive Message Cancellation

Cancellation of an asynchronous receive request simply removes that request from the port. The receive request, by definition, has not been matched with a message, so no special action is required. The receive request is completed with a status that indicates the cancellation. The receiver must check the status to distinguish a receive that is being canceled from a message that is being received.

```
OSStatus CancelAsyncReceive (ReceiveID theReceive);
```

theReceive is the ID of a pending receive operation to be canceled.

Client Initiated Cancellation Messages

Servers must be prepared to receive cancellation messages in case their clients decide to cancel requests. These messages are generated by the microkernel in response to calls to the CancelMessageSend service. Cancellation messages, like any other kernel message, begin with a header that describes the kernel message. The remainder of the message contains the ID of the message being canceled and a reason for cancellation. The reason is a 32-bit value that is not used by the microkernel. Its meaning is part of the client-server interface.

```
enum
{
    kCancelMessageCode = 3
};

typedef struct CancelMessage
{
    KernelMessageHeader header;
```

```
    MessageID      cancelledMessage;  
    OSStatus       reason;  
} CancelMessage;
```

GETTING SYSTEM INFORMATION

The `GetSystemInformation` system call returns various pieces of information about the microkernel and the CPU. It's the kind of information that would generally go into the system registry or Gestalt. In order to avoid having the microkernel depend on those modules, a system call is provided instead.

```
enum
{
    kSystemInformationVersion = 1
};

typedef struct SystemInformation
{
    ItemCount          numPhysicalRAMPages;
    ItemCount          numFreeRAMPages;
    ItemCount          numEligibleRAMPages;
    ItemCount          numResidentRAMPages;
    ItemCount          numInMemoryGlobalPages;
    ItemCount          numLogicalPages;
    ByteCount          pageSize;
    ByteCount          dataCacheBlockSize;
    UInt32             processorVersionNumber;
    UInt32             numCPUs;
    KernelProcessID    systemKernelProcessID;
    AddressSpaceID     globalAddressSpaceID;
};

OSStatus GetSystemInformation (PBVersion          theVersion,
                              SystemInformation * theSystemInfo)
```

`GetSystemInformation` returns information about the CPU and microkernel.

`theVersion` specifies the version number of `SystemInformation` to be returned. This provides backward compatibility. `kSystemInformationVersion` is the version of `SystemInformation` defined in the current interface.

`theSystemInfo` specifies where to return the information.

The fields of `SystemInformation` are:

- `numPhysicalRAMPages` - the total number of physical RAM pages in the system. This number doesn't include memory used for video frame buffers.
- `numFreeRAMPages` - the number of physical memory pages available for use by the kernel without any backing object activity.
- `numEligibleRAMPages` - the number of pageable physical memory pages that are eligible for replacement, but require backing object activity to obtain.

- numResidentRAMPages - the number of nonpageable physical memory pages. This includes pages made resident by ControlPagingForRange or PrepareMemoryForIO, and those in resident areas.

Note: Locked physical pages mapped more than once into a given address space are counted more than once in numResidentRAMPages.

- numInMemoryGlobalPages - the number of pageable pages in global areas currently in memory.
- numLogicalPages - the cumulative size of all areas into which physical memory pages can be mapped, exclusive of guard pages. $V/R = \text{numLogicalPages} / \text{numPhysicalRAMPages}$.

Note: Areas that share physical pages are accumulated just like any other areas, even though their need for memory is less than the sum of their sizes.

- pageSize - the page size used by the CPU. Most virtual memory operations are applied only to entire pages. This value may not always be the same on the same type of CPU. Do not assume that PowerPC pages are 4K.
- dataCacheBlockSize - the cache line size affected by CPU cache control instructions (for example, dcbf on PowerPC). Only libraries providing cache manipulation routines should use this value or the instructions that depend on it.
- processorVersionNumber - the version number returned from the CPU's "read processor version number" instruction, if any.
- numCPUs - the number of CPUs in the system. This doesn't count coprocessors such as DSPs or CPUs managed by software other than the microkernel.
- systemKernelProcessID - the ID of the kernel process to which all privileged tasks belong.
- globalAddressSpaceID - the ID of the address space to which all global areas belong.

RESTRICTIONS ON USING MICROKERNEL SERVICES

The microkernel provides three separate execution levels: task level, hardware interrupt level, and secondary interrupt level. Additionally, task-level execution consists of execution of privileged tasks and non-privileged tasks. Various restrictions are placed on which microkernel services can be called from each of these execution levels. The following sections define which services are available from each execution level.

Services That Can Be Called From Task Level

All microkernel services may be called from task level. However, not all microkernel services are available to non-privileged tasks. These services are listed in the following section.

Services That Cannot Be Called By Non-Privileged Tasks

- CallSecondaryInterruptHandler2
- QueueSecondaryInterruptHandler
- AcceptMessage

Services That Can Be Called From Secondary Interrupt Handlers

- SetEvents
- ClearEvents
- ReadEvents
- NotifyKernelQueue
- RemainingStackSize
- GetTaskInformation
- SetTaskPriority
- CreateSoftwareInterrupt
- SendSoftwareInterrupt
- DeleteSoftwareInterrupt

- CallSecondaryInterruptHandler2
- QueueSecondaryInterruptHandler
- SetInterruptTimer
- CancelTimer
- SetProcessorCacheMode
- CheckpointIO
- GetRegistryObjectID
- SetRegistryObjectID
- ReplyToMessage

Services That Can Be Called From Hardware Interrupt Level

The following services which are callable from hardware interrupt level are also the only services callable with hardware interrupts disabled.

- SetEvents
- ClearEvents
- SendSoftwareInterrupt
- QueueSecondaryInterruptHandler
- SetInterruptTimer