INTEGRATED SYSTEMS, INC.

pSOSystem

System Concepts

INTEGRATED SYSTEMS

pSOSystem

SYSTEM CONCEPTS

Release 2.0

Integrated Systems, Inc.
3260 Jay Street
Santa Clara, CA. 95054

Phone: (408) 980-1500 Telex: 757697 (soft com)
Fax: (408) 980-0400 Email: scg_suprt@isi.com

Document Title: **pSOSystem System Concepts**
Document Number: **PS2-000-003**
Part Number: **PS2000MAN**
Revision Date: **6 December 1993**

Any questions or comments on this manual are welcome. Please fax them to the number listed above, or send them to Technical Publications at the above address.

Preface

1 Product Overview

2 pSOS+ Real-Time Kernel

3 pSOS+m Multiprocessing Kernel

4 Network Programming

List of Figures

Preface

Purpose

This manual is part of a documentation set that describes pSOSystem, the modular, high-performance real-time operating system environment from Integrated Systems.

This manual provides theoretical information about the operation of pSOSystem. The level of information in this manual is different than that provided by either the *pSOSystem Getting Started* manual or the *pSOSystem Programmer's Reference.* Together, the three manuals comprise the basic documentation set for pSOSystem.

Read this manual to gain an understanding of how the various software components in pSOSystem can be combined to create an environment suited to your particular needs.

Audience

This manual is targeted primarily for embedded application developers who want to gain an overall understanding of pSOSystem components. Basic familiarity with UNIX terms and concepts is assumed.

A secondary audience includes those seeking an introduction to pSOSystem features.

Organization

This manual is organized as follows:

Chapter 1, Product Overview, presents a brief introduction to pSOSystem and its core components.

Chapter 2, pSOS+ Real-Time Kernel, describes the pSOS+ real-time multitasking kernel, the heart of pSOSystem.

Chapter 3, pSOS+m Multiprocessing Kernel, describes the extensions offered by the pSOS+m multitasking, multiprocessing kernel.

Chapter 4, Network Programming, provides a summary of pSOSystem's networking services and describes in detail the pNA+ TCP/IP Manager component.

Chapter 5, pHILE+ File System Manager, describes pSOSystem's file management option.

Chapter 6, pREPC+ ANSI C Library, describes pSOSystem's ANSI C run-time library.

Chapter 7, I/O System, discusses the pSOSystem I/O system and explains how device drivers are incorporated into a system.

Related Documentation

When using pSOSystem you might want to have on hand the other two manuals included in the basic documentation set:

- *pSOSystem Getting Started* - explains how to create and bring up pSOSystem-based applications. This manual also contains installation instructions and a number of tutorials.
- *pSOSystem Programmer's Reference* - contains detailed descriptions of all pSOSystem system calls and system services, as well as other important reference material such as error codes, configuration tables, and memory usage.

Based on the options you have purchased, you might also need to reference one or more of the following manuals:

- *OpEN User's Manual* - describes how to install and use pSOSystem's OpEN (Open Protocol Embedded Networking) product.

- *SNMP User's Manual* - describes the internal structure and operation of SNMP, Integrated System's Simple Network Management Protocol product. This manual also describes how to install and use the SNMP MIB (Management Information Base) Compiler.
- *pROBE+ User's Manual* - describes how to use the pROBE+ System Debugger/Analyzer.
- *pSOSim User's Manual* - describes how to install and use pSOSim, a UNIX-based pSOS+ kernel simulator.
- *XRAY+ User's Manual* - describes how to use the XRAY+ Source-Level Cross Debugger.

Notation Conventions

The following notation conventions are used in this manual:

- Function names (**q_receive**), filenames (**pdefs.h**), keywords (**int**), and operators (**!**) that must be typed exactly as shown are presented in bold.
- Italics indicate that a user-defined value or name (*drive:pathname*) can be substituted for the italicized word(s) shown. Italics also indicate emphasis, such as when important terms are introduced.
- Keynames [Enter] are shown within square brackets. Keynames separated by hyphens are typed together. For example, to type [Ctrl-Shift-E], hold down the [Ctrl] and [Shift] keys and type the letter E.
- `Code examples are shown in constant width.`

(Blank Page)

1 Product Overview

1.1 What Is pSOSystem?

pSOSystem is a modular, high-performance real-time operating system designed specifically for embedded microprocessors. It provides a very complete, multitasking environment based on open systems standards.

pSOSystem is designed to meet three overriding objectives:

- Performance
- Reliability
- Ease-of-Use

The result is a fast, deterministic, robust, yet accessible, system software solution. Accessible in this case translates to a minimal learning curve. pSOSystem is designed for quick startup on both custom and commercial hardware.

pSOSystem is supported by an integrated set of cross development tools that can reside on UNIX- or DOS-based computers. These tools can communicate with a target over a serial or TCP/IP network connection.

1.2 System Architecture

pSOSystem employs a modular architecture. It is built around the pSOS+ real-time multi-tasking kernel and a collection of companion software components. Software components are standard building blocks delivered as absolute position-independent code modules. They are "standard parts" in the sense that they are unchanged from one application to another. This black box technique eliminates maintenance by the user and assures reliability, since hundreds of applications execute the same, identical code.

Unlike most system software, a software component is not wired down to a piece of hardware. It makes no assumptions about the execution/target environment. Each software component utilizes a user-supplied configuration table that contains application- and hardware-related parameters to configure itself at startup.

Every component implements a logical collection of system calls. To the application developer, system calls appear as re-entrant C functions callable from an application. Any combination of components can be incorporated into a system to match your real-time design requirements. pSOSystem includes the following components:

- **pSOS+ Real-time Multitasking Kernel.** A field-proven, multitasking kernel that provides a responsive, efficient mechanism for coordinating the activities of your real-time system.
- **pSOS+m Multiprocessor Multitasking Kernel.** Extends the pSOS+ feature set to operate seamlessly across multiple, tightly-coupled or distributed processors.
- **pNA+ TCP/IP Network Manager.** A complete TCP/IP implementation including gateway routing, UDP, ARP, and ICMP protocols; uses a standard socket interface that includes stream, datagram, and raw sockets.
- **pRPC+ Remote Procedure Call Library.** Offers SUN-compatible RPC and XDR services; allows you to build distributed applications using the familiar C procedure paradigm.
- **pHILE+ File System Manager.** Gives efficient access to mass storage devices, both local and on a network. Includes support for MS-DOS compatible floppy disks and a high-speed proprietary file system. When used in conjunction with pNA+ and pRPC+, offers client-side NFS services.

- **pREPC+ ANSI C Standard Library**. Provides familiar ANSI C run-time functions such as **printf()**, **scanf()**, and so forth, in the target environment.
- **pX11+ X Window System**. Allows pSOSystem machines to display output and receive input from any X server on a network reachable via pNA+.

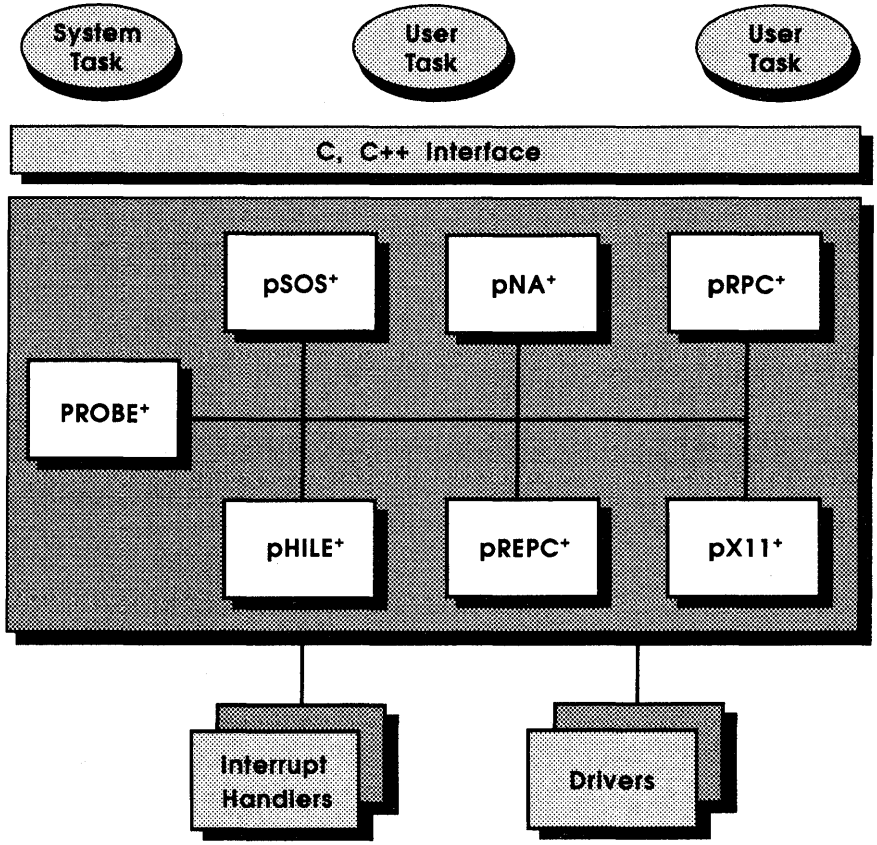

Figure 1. The pSOSystem Environment

In addition to these core components, pSOSystem includes the following:

- Networking protocols including SNMP, FTP, Telnet, TFTP, NFS, and STREAMS
- Run-time loader
- User application shell
- Support for C++ applications
- Boot ROMs
- Pre-configured versions of pSOSystem for popular commercial hardware
- pSOSystem templates for custom configurations
- Chip-level device drivers
- Sample applications

This manual focuses on explaining pSOSystem core components. Other parts of pSOSystem are described in the *pSOSystem Programmer's Reference* and in the *pSOSystem Getting Started* manual.

1.3 Integrated Development Environment

pSOSystem's integrated cross development environment can reside on a UNIX- or DOS-based computer. It includes C and C++ optimizing compilers, a target cpu simulator, a pSOS+ OS simulator, and a cross-debug solution that supports source- and system-level debugging.

pSOSystem's debugging environment centers on the XRAY+ Source-Level Debugger and the pROBE+ System-Level Debugger. XRAY+ executes on your host computer and works in conjunction with pROBE+, which runs on a target system.

The XRAY+/pROBE+ combination provides a multitasking debug solution that features:

- A sophisticated mouse and window user interface
- Automatic tracking of program execution through source code files
- Traces and breaks on high-level language statements
- Breaks on task state changes and operating system calls

- Monitoring of language variables and system-level objects such as tasks, queues and semaphores
- Profiling for performance tuning and analysis
- System and task debug modes
- The ability to debug optimized code

pROBE+, in addition to acting as a "back end" for XRAY+, can function as a standalone target-resident debugger that can be delivered with a user's final product to provide a field maintenance capability.

XRAY+, pROBE+, and pSOSystem's other development tools are described in other manuals. See "Related Documentation" in the Preface to this manual.

(Blank Page)

2 pSOS+ Real-Time Kernel

2.1 Overview

pSOS+ is a real-time, multitasking operating system kernel. As such, it acts as a nucleus of supervisory software that

- Performs services on demand;
- Schedules, manages, and allocates resources;
- Generally coordinates multiple, asynchronous activities.

pSOS+ maintains a highly simplified view of application software, irrespective of the application's inner complexities. To the pSOS+ kernel, applications consist of three classes of program elements:

- Tasks
- I/O Device Drivers
- Interrupt Service Routines (ISRs)

Tasks, their virtual environment, and ISRs are the primary topics of discussion in this chapter. The I/O system and device drivers are discussed in Chapter 7, "I/O System."

Discussions in this section focus primarily on concepts relevant to a single-processor system. Additional issues and considerations introduced by multiprocessor configurations are covered in Chapter 3, "pSOS+m Multiprocessing Kernel."

2.2 The Real-Time Design Problem

A so-called real-time system is characterized, above and beyond a certain set of quantifiable time imperatives, by its need to act upon

- Multiple asynchronous, external events,
- Multiple cyclical, external events, or
- A combination of the above.

Yet, software is by its nature synchronous: one instruction follows another. How then to handle these externally clocked, or worse, asynchronous, events?

The traditional method uses interrupt handlers and one all-encompassing control loop; an example is shown in Figure 2. The interrupt handlers react to external events, then log the events using flags and/or data buffers. Meanwhile, in the background, the synchronous control loop successively tests the flags or feeds the buffers, and performs whatever processing is called for.

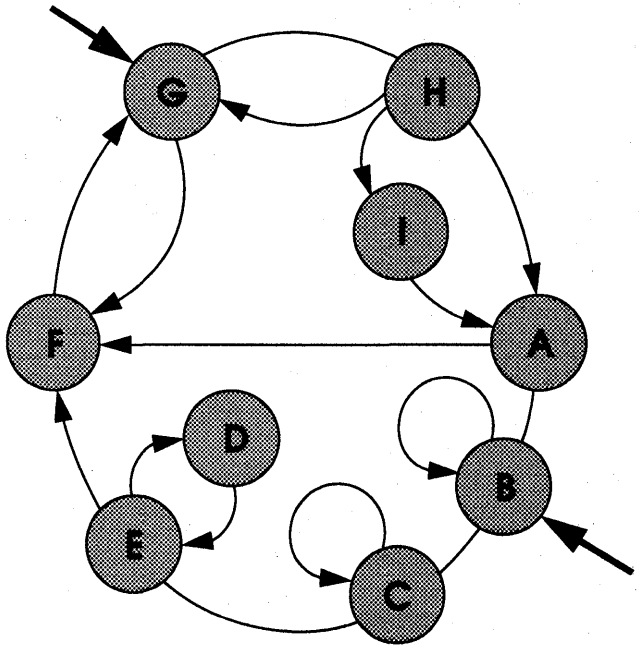

Figure 2. Ad Hoc Software Design

On the surface, this is a viable solution. It suffers, however, from serious, intrinsic flaws, due mainly to the single-threaded nature of the control loop.

For one thing, in a simple control loop, the worst case response time to a given event is equal to the time it takes the loop to go around once. Plus, there is no allowance for prioritization of event processing, whose order is dictated instead by the static sequence built into the control loop.

True, this behavior can be modified by conditionally altering the test sequence within the control loop. For example, after processing event A, if event X is present and event Y has been tasked, then do task event X, else go on to test for event B; but it is easy to see that this can quickly get out of hand.

A more serious flaw has to do with the non-preemptible nature of a synchronous loop. Once the loop starts processing event A, it must finish this action, no matter how long it takes, before it will have any chance to decide which event to poll or task next.

Picture a train winding through a series of tunnels. Once it enters a tunnel, it will not see the light of day again until it exits the tunnel; but then comes the next tunnel. In a similar manner, a single, long computational action affects the entire system's response.

This result can be ameliorated by sectioning each long tunnel into a series of short ones; or, more processing can be shifted to the interrupt handlers. But the requisite pre-analysis and state tracking will add further to the bewildering considerations a designer must address and overcome.

And, of course, there are ubiquitous race conditions to contend with. Data buffers, queues, flags, and other structures accessible by multiple sources, notably interrupt handlers, must be locked with critical regions to insure atomic operation. This is never an easy task. One oversight, and a latent race condition will lie in wait.

These problems associated with ad hoc software design should be familiar to most real-time software engineers. It is a very costly proposition for designers to have to tackle them anew on every new project or product implementation.

2.3 Multitasking Implementation

The pSOS+ multitasking kernel, illustrated in Figure 3, provides an alternative to the ad hoc method. In place of interrupt handlers coupled by polling to a synchronous control loop, pSOS+ allows interrupt handlers to directly trigger tasks that can in turn trigger other tasks.

There are two intrinsic differences between control loop and multitasking implementations. First, a control loop is synchronous. (One action must complete before another one can start.) In a multitasking system, while tasks are internally synchronous, different tasks can execute asynchronously. A task can also be stopped in mid-stream, and execution passed to another task at any time.

Second, a control loop is static -- its execution sequence is highly dependent on the way it was programmed. By contrast, a multitasked system is dynamic, since task switching is basically driven by temporal events.

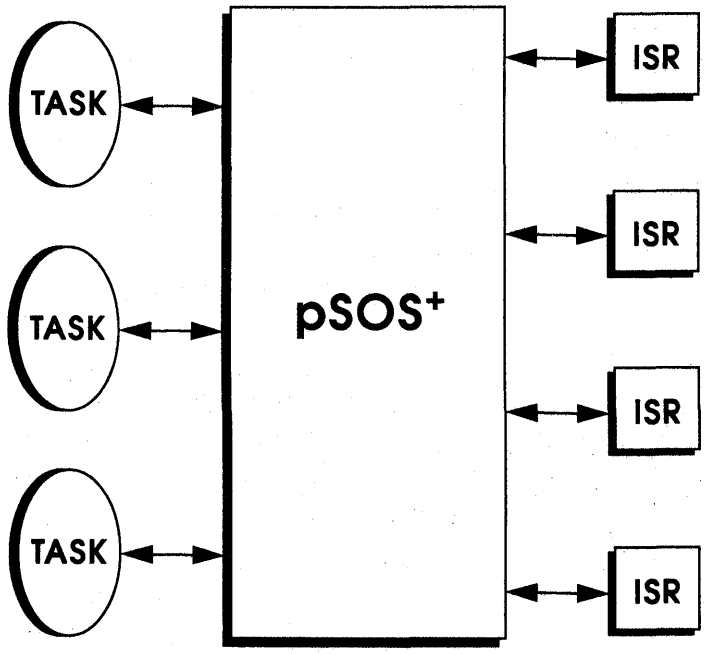

Figure 3. Multitasking Approach

It is easy to see that a multitasked implementation is a much more natural match to the outside world, which is mainly asynchronous and/or cyclical as far as real-time systems are concerned. Thus, application software developed for multitasking systems is likely to be far more structured, race-free, maintainable, and re-usable.

Several pSOS+ attributes help solve the problems inherent in real-time software development. They include

- Partitioning of actions into multiple tasks, each capable of executing in parallel (i.e., overlapping) with other tasks. The pSOS+ kernel switches on cue between tasks, thus enabling applications to act asynchronously -- in tune with the outside world.
- Task prioritization. pSOS+ always executes the highest priority task that can run.
- Task preemption. If an action is in progress and a higher priority external event occurs, the event's associated action takes over immediately.
- Powerful, race-free synchronization mechanisms available to applications. These include message queues, semaphores, multiple-wait events, and asynchronous signals.
- Timing functions, such as wakeup, alarm timers, and timeouts, for servicing cyclical, external events.

2.3.1 Concept of a Task

From the system's perspective, a task is the smallest unit of execution that can compete on its own for system resources. A task lives in a virtual, insulated environment furnished by the pSOS+ kernel. Within this space, a task can use system resources or wait for them to become available, if necessary, without explicit concern for other tasks. Resources include the CPU, I/O devices, memory space, and so on.

Conceptually, a task can execute concurrently with, and independently of, other tasks. The pSOS+ kernel simply switches between different tasks on cue. The cues come by way of system calls to pSOS+. For example, a system call might cause the kernel to stop one task in mid-stream and continue another from the last stopping point.

Although each task is a logically separate set of actions, it must coordinate and synchronize itself, with actions in other tasks or with ISRs, by calling pSOS+ system services.

2.3.2 Decomposition Criteria

The decomposition of a complex application into a set of tasks and ISRs is a matter of balance and trade-offs, but one which obviously impacts the degree of parallelism, and therefore efficiency, that can be achieved. Excessive decomposition exacts an inordinate amount of overhead activity required in switching between the virtual environments of different tasks. Insufficient decomposition reduces throughput, since actions in each task proceed serially, whether they need to or not.

There are no fixed rules for partitioning an application; the strategy used depends on the nature of the application. First of all, if an application involves multiple, independent main jobs (for example, control of N independent robots), then each job should have one or more tasks to itself. Within each job, however, the partitioning into multiple, cooperating tasks requires much more analysis and experience.

The following discussion presents a set of reasonably sufficient criteria, whereby a job with multiple actions can be divided into separate tasks. Note that there are no necessary conditions for combining two tasks into one task, though this might result in a loss of efficiency or clarity. By the same token, a task can always be split into two, though perhaps also with some loss of efficiency.

Terminology:

In this discussion, a *job* is defined as a group of one or more tasks, and a *task* is defined as a group of one or more actions.

An action (*act*) is a locus of instruction execution, often a loop.

A dependent action (*dact*) is an action containing one and only one dependent condition; this condition requires the action to wait until the condition is true, but the condition can only be made true by another dact.

Decomposition Criteria:

Given a task with actions A and B, if any one of the following criteria are satisfied, then actions A and B should be in separate tasks:

Time -- dact A and dact B are dependent on cyclical conditions that have different frequencies or phases.

Asynchrony -- dact A and dact B are dependent on conditions that have no temporal relationships to each other.

Priority -- dact A and dact B are dependent on conditions that require a different priority of attention.

Clarity/Maintainability -- act A and act B are either functionally or logically removed from each other.

pSOS+ imposes essentially no limit on the number of tasks that can coexist in an application. You simply specify in the pSOS+ Configuration Table the maximum number of tasks expected to be active contemporaneously, and pSOS+ allocates sufficient memory for the requisite system data structures.

2.4 Overview of System Operations

pSOS+ services can be separated into the following categories:

- Task Management
- Storage Allocation
- Message Queue Services
- Event and Asynchronous Signal Services
- Semaphore Services
- Time Management and Timer Services
- Interrupt Completion Service
- Error Handling Service
- Multiprocessor Support Services

Detailed descriptions of each system service call are provided in the *pSOSystem Programmer's Reference* manual. The remainder of this chapter, which provides more details on the principles of pSOS+ operation, is highly recommended reading for first-time pSOS+ users.

2.4.1 Task States

A task can be in one of several execution states. A task's state can change only as result of a system call made to pSOS+ by the task itself, or by another task or ISR. From a macroscopic perspective, a multitasked application moves along by virtue of system calls into pSOS+, forcing pSOS+ to then change the states of affected tasks and, possibly as a result, switch from running one task to running another.

Therefore, gaining a complete understanding of task states and state transitions is an important step towards using pSOS+ properly and fully in the design of multitasked applications.

To pSOS+, a task simply does not exist either before it is created, or after it is deleted. A created task must be started before it can execute. A *created-but-unstarted* task is therefore in an innocuous, embryonic state.

Once *started*, a task resides in one of three major states:

1. Ready
2. Running
3. Blocked

A ready task is runnable (not blocked), and waits only for higher priority tasks to release the CPU. Since a task can be started only by a call from a running task, and there can be only one running task at any given instant, a new task always starts in the ready state.

A running task is simply a ready task that has been given use of the CPU. There is always one and only one running task. In general, the running task has the highest priority among all ready tasks; however, there are a few user-selectable exceptions.

A task becomes blocked only as the result of some deliberate action on the part of the task itself, usually a system call that causes the calling task to wait. Thus, a task cannot go from the ready state to blocked, because only a running task can send system calls.

2.4.2 State Transitions

Figure 4 depicts the possible states and state transitions for a pSOS+ task. Each state transition is described in detail below. Note the following abbreviations:

- E for Running (Executing)
- R for Ready
- B for Blocked

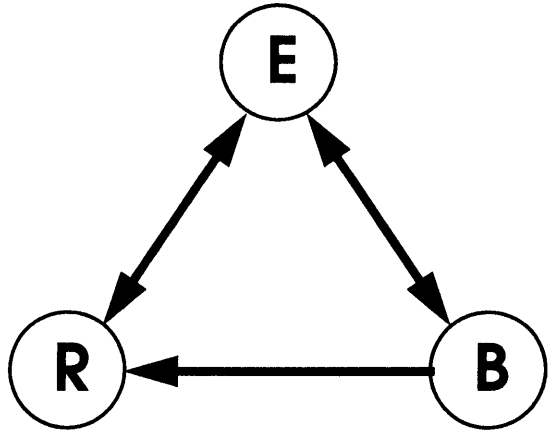

Figure 4. Task State Transitions

(E->B) A running task (E) becomes blocked when:

1. It requests a message (**q_receive/q_vreceive** with wait) from an empty message queue; or
2. It waits for an event condition (**ev_receive**) that is not presently pending; or
3. It requests a semaphore token (**sm_p** with wait) that is not presently available; or
4. It requests memory (**rn_getseg** with wait) that is not presently available; or
5. It pauses for a time interval (**tm_wkafter**) or until a particular time (**tm_wkwhen**).

(B->R) A blocked task (B) becomes ready when:

1. A message arrives at the message queue (**q_send/q_vsend**, **q_urgent/q_vurgent**, **q_broadcast/q_vbroadcast**) where B has been waiting, and B is first in that wait queue; or
2. An event is sent to B (**ev_send**), fulfilling the event condition it has been waiting for; or
3. A semaphore token is returned (**sm_v**), and B is first in that wait queue; or

4. Memory returned to pSOS+ (**rn_retseg**) now allows a memory segment that has been waiting in the memory region's wait queue to be allocated to B; or
5. B has been waiting with a timeout option for events, a message, a semaphore, or a memory segment, and that timeout interval expires; or
6. B has been delayed, and its delay interval expires or its wakeup time arrives; or
7. B is waiting at a message queue, semaphore or memory region, and that queue, semaphore or region is deleted by another task.

(B->E) A blocked task (B) becomes the running task when:

1. Any one of the (B->R) conditions occurs, and B also has higher priority than the last running task, provided that task has preemption enabled.

(R->E) A ready task (R) becomes running when the last running task (E):

1. Blocks; or
2. Re-enables preemption, and R has higher priority than E; or
3. Has preemption enabled, and E changes its own, or R's, priority so that R now has higher priority than E and all other ready tasks; or
4. Runs out of its timeslice, its roundrobin mode is enabled, and R has the same priority as E.

(E->R) The running task (E) becomes a ready task when:

1. Any one of the (B->E) conditions occurs for a blocked task (B) as a result of a system call by E or an ISR; or
2. Any one of the conditions 2-4 of (R->E) occurs.

A fourth, but secondary, state is the suspended state. A suspended task simply cannot run, until it is explicitly resumed. Suspended is very similar to blocked, but there are fundamental differences.

First, a task can only block itself, but it can suspend other tasks as well as itself. One result is that a ready task can be removed from the ready state by suspension.

Second, a blocked task can also be suspended. In this case, the effects are additive -- that task must be both unblocked and resumed, the order being irrelevant, before the task can become ready or running.

NOTE: The task states discussed above should not be confused with user and supervisor program states that exist in MC68000-based systems. The latter are hardware states of privilege. A task can run in the user state, or the supervisor state, or switch at will between the two -- its hardware state has nothing to do with its task state.

2.4.3 Task Scheduling

pSOS+ employs a priority-based, preemptive scheduling algorithm. In general, pSOS+ ensures that, at any point in time, the running task is the one with the highest priority among all ready-to-run tasks in the system. However, you can modify pSOS+ scheduling behavior by selectively enabling and disabling preemption or time-slicing for one or more tasks.

Each task has a mode word (see section 2.5.3, "Task Mode Word"), with two settable bits that can affect scheduling. One bit controls the task's preemptibility. If disabled, then once the task enters the running state, it will stay running even if other tasks of higher priority enter the ready state. A task switch will occur only if the running task blocks, or if it re-enables preemption.

A second mode bit controls timeslicing. If the running task's timeslice bit is enabled, pSOS+ automatically tracks how long the task has been running. When the task exceeds the predetermined timeslice, and other tasks with the same priority are ready to run, pSOS+ switches to run one of those tasks. Timeslicing only affects scheduling among equal priority tasks. For more details on timeslicing, see section 2.4.5, "Roundrobin by Timeslicing."

2.4.4 Task Priority

A priority must be assigned to each task when it is created. There are 256 priority levels -- 255 is the highest, 0 the lowest. Certain priority levels are reserved for use by special pSOSystem tasks. Level 0 is reserved for the **IDLE** daemon task furnished by pSOS+. Levels 240 - 255 are reserved for a variety of high priority pSOSystem tasks, including the pSOS+ **ROOT** task, which runs at level 240. A task's priority, including that of system tasks, can be changed at runtime by calling the **t_setpri** system call.

When a task enters the ready state, pSOS+ puts it into an indexed ready queue, behind tasks of higher or equal priority. As noted earlier, all ready queue operations, including insertions and removals are achieved in fast, constant time. No search loop is ever needed.

During dispatch, when it is about to exit and return to the application code, pSOS+ will normally run the task at the top of the ready queue. If this is the same task that was last running, then pSOS+ simply returns to it. Otherwise, the last running task must have either blocked, or one or more ready tasks now have higher priority. In the first (blocked) case, pSOS+ will always switch to run the task currently at the top of the indexed ready queue. In the second case, technically known as preemption, pSOS+ will also perform a task switch, unless the last running task has its preemption mode disabled, in which case the dispatcher has no choice but to return to it.

Note that a running task can only be preempted by a task of higher or equal (if timeslicing enabled) priority. It should be obvious that assignment of priority levels is crucial in any application. A particular ready task simply cannot run, unless all tasks with higher priority are blocked. By the same token, a running task can be preempted at any time, if an interrupt occurs and the attendant ISR unblocks a higher priority task.

2.4.5 Roundrobin by Timeslicing

In addition to priority, pSOS+ can use timeslicing to schedule task execution. However, timesliced (roundrobin) scheduling can be turned on/off on a per task basis, and is always secondary to priority considerations.

You can specify the timeslice quantum in the Configuration Table using the parameter **kc_ticks2slice**. For example, if this value is 6, and the clock frequency (**kc_ticks2sec**) is 60, a full slice will be 1/10 second.

Each task carries a timeslice counter, initialized by pSOS+ to the timeslice quantum when the task is created. Whenever a clock tick is announced to pSOS+, the pSOS+ time manager checks the running task's mode bits. If the task's roundrobin bit or the preemption bit are disabled, then its timeslice counter is untouched and ignored. If the running task's roundrobin bit and preemption bit are enabled, then

pSOS+ decrements its timeslice counter, unless it is already zero. If the count is 0, there are two possible outcomes, as follows:

1. If all other presently ready tasks have lower priority, then no special scheduling takes place. The task's timeslice counter stays at zero, so long as it stays in the running or ready state.
2. If one or more other tasks of the same priority are ready, pSOS+ moves the running task from the running state into the ready state, and re-enters it into the indexed ready queue behind all other ready tasks of the same priority. This forces the pSOS+ dispatcher to switch from that last running task to the task now at the top of the ready queue. The last running task's timeslice counter is given a full timeslice, in preparation for its next turn to run.

Regardless of whether or not its roundrobin mode bit is enabled, when a task becomes ready from the blocked state, pSOS+ always inserts it into the indexed ready queue behind all tasks of higher or equal priority. At the same time, the task's timeslice counter is refreshed with a new, full count.

NOTE: The preemption mode bit takes precedence over roundrobin scheduling. If the running task has preemption disabled, then it will preclude roundrobin and continue to run.

In general, real-time systems rarely require time-slicing, except to insure that certain tasks will not inadvertently monopolize the CPU. Therefore, pSOS+ always initializes each task with the roundrobin mode disabled. For certain applications, automatic roundrobin based on timeslice might not be desirable.

For example, shared priority is often used to prevent mutual preemption among certain tasks, such as those that share non-reentrant critical regions. In such cases, roundrobin should be left disabled for all such related tasks, in order to prevent pSOS+ from switching tasks in the midst of such a region.

To maximize efficiency, a task's roundrobin should be left disabled, if:

1. it has a priority level to itself, or
2. it shares its priority level with one or more other tasks, but roundrobin by timeslice among them is not necessary.

2.4.6 Manual Roundrobin

As noted earlier, automatic roundrobin by timeslice might not be suitable for certain applications. However, there might still be a need to perform roundrobin manually -- that is, the running task might need to explicitly give up the CPU to other ready tasks of the same priority.

pSOS+ supports manual roundrobin, via the **tm_wkafter** system call with a zero interval. If the running task is the only ready task at that priority level, then the call simply returns to it. If there are one or more ready tasks at the same priority, then pSOS+ will take the calling task from the running into the ready state, thereby putting it behind all ready tasks of that priority. This forces pSOS+ to switch from that last running task to another task of the same priority now at the head of the ready queue.

2.4.7 Dispatch Criteria

Dispatch refers to the exit stage of pSOS+, where it must decide which task to run upon exit; that is, whether it should continue with the running task, or switch to run another ready task.

If pSOS+ is entered because of a system call from a task, then pSOS+ will definitely exit through the dispatcher, in order to catch up with any state transitions that might have been caused by the system call. For example, the calling task might have blocked itself, or made a higher priority blocked task ready. On the other hand, if pSOS+ is entered because of a system call by an ISR, then pSOS+ will not dispatch, but will instead return directly to the calling ISR, to allow the ISR to finish its duties.

Since a system call from an ISR might have caused a state transition, such as readying a blocked task, a dispatch must be forced at some point. This is the reason for the **I_RETURN** entry into pSOS+, which is used by an ISR to exit the interrupt service, and at the same time allow pSOS+ to execute a dispatch.

2.4.8 Objects, Names, and IDs

pSOS+ is an object-oriented operating system kernel. Object classes include tasks, memory regions, memory partitions, message queues, and semaphores.

Each object is created at runtime and known throughout the system by two identities -- a pre-assigned name and a run-time ID. An object's 32-bit (4 characters, if ASCII) name is user-assigned and passed to pSOS+ as input to an **Obj_CREATE** (e.g. **t_create**) system call. pSOS+ in turn generates and assigns a unique, 32-bit object ID (e.g. **Tid**) to the new object. Except for **Obj_IDENT** (e.g. **q_ident**) calls, all other system calls that reference an object must use its ID. For example, a task is suspended using its **Tid**, a message is sent to a message queue using its **Qid**, and so forth.

The run-time ID of an object is of course known to its creator task -- it is returned by the **Obj_CREATE** system call. Any other task that knows an object only by its user-assigned name can obtain its ID in one of two ways:

1. Use the system call **Obj_IDENT** once with the object's name as input; pSOS+ returns the object's ID, which can then be saved away.
2. Or, the object ID can be obtained from the parent task in one of several ways. For example, the parent can store away the object's ID in a global variable -- the **Tid** for task **ABCD** can be saved in a global variable with a name like **ABCD_TID**, for access by all other tasks.

An object's ID contains implicitly the location, even in a multiprocessor distributed system, of the object's control block (e.g. TCB or QCB), a structure used by pSOS+ to manage and operate on the abstract object. This is an important notion, since using the ID to reference an object eliminates the need for pSOS+ to search for its control structure.

Objects are truly dynamic -- the binding of a named object to its reference handle is deferred to runtime. By analogy, pSOS+ treats objects like files. A file is created by name. But to avoid searching, read and write operations use the file's ID returned by create or open. Thus, **t_create** is analogous to **File_Create**, and **t_ident** to **File_Open**.

As noted above, an object's name can be any 32-bit integer. However, it is customary to use four-character ASCII names, since ASCII names are more easily remembered, and pSOSystem debug tools will display an object name in ASCII, if possible.

2.5 Task Management

In general, task management provides dynamic creation and deletion of tasks, and control over task attributes. The available system calls in this group are:

t_create	Create a new task.
t_ident	Get the ID of a task.
t_start	Start a new task.
t_restart	Restart a task.
t_delete	Delete a task.
t_suspend	Suspend a task.
t_resume	Resume a suspended task.
t_setpri	Change a task's priority.
t_mode	Change calling task's mode bits.
t_setreg	Set a task's notepad register.
t_getreg	Get a task's notepad register.

2.5.1 Birth of a Task

Task creation refers to operations that pass a task and its attributes to the pSOS+ kernel, so that pSOS+ can schedule the task for execution and allow it to compete for other system resources.

The code segment of a task must be memory resident. It can be in ROM, or loaded into RAM either at startup or at the time of its creation. A task's data area can be statically assigned, or dynamically requested from pSOS+. Memory considerations are discussed in detail in the *pSOSystem Programmer's Reference* manual.

Task creation requires two system calls -- **t_create** and **t_start**. A parent task creates an offspring task by calling **t_create**, and passing the following as input parameters:

- A user-assigned name
- A priority level for scheduling purposes
- Sizes for one or two stacks
- Several flags

One flag is meaningful only in a multiprocessor system (see Chapter 3, "pSOS+m Multiprocessing Kernel"). A second flag specifies whether the task uses the FPU coprocessor.

t_create acquires and sets up a Task Control Block (TCB) for the newcomer, then it allocates a memory segment (from Region 0) large enough for the task's stack(s) and any necessary extensions. Extensions are extra memory areas required for optional features. For example:

- An FPU save area for systems with co-processors
- Memory needed by other system components (such as pHILE+, pREPC+, pNA+, and so forth) to hold per-task data

This memory segment is linked to the TCB. **t_create** returns a task identifier assigned by pSOS+.

The **t_start** call must be used to complete the creation. **t_start** supplies the starting address of the new task, a mode word that controls its initial execution behavior (see section 2.5.3, "Task Mode Word"), and an optional argument list. Once started, the task is ready-to-run, and is scheduled for execution based on its assigned priority.

With two exceptions, all user tasks that form a multitasking application are created dynamically at runtime. One exception is the **ROOT** task, which is created and started by pSOS+ as part of its startup initialization. After startup, pSOS+ simply passes control to the **ROOT** task. The other exception is the default **IDLE** task, also provided as part of startup. All other tasks are created by explicit system calls to pSOS+, when and as needed.

In some designs, **ROOT** can simply initialize the rest of the application by creating at once all the other tasks. In other systems, **ROOT** might create a few tasks, which in turn can create a second layer of tasks, which in turn can create a third layer, and so on. The total number of active tasks in your system is limited by the **KC_NTASK** specification in the pSOS+ Configuration Table.

2.5.2 Task Control Block

A task control block (TCB) is a system data structure allocated and maintained by pSOS+ for each task after it has been created. A TCB contains everything the kernel needs to know about a task, including its name, priority, remainder of timeslice, and of course its context. Generally, context refers to the state of machine registers. When a task

is running, its context is highly dynamic and is the actual contents of these registers. When the task is not running, its context is frozen and kept in the TCB, to be restored the next time it runs.

There are certain overhead structures within a TCB that are used by pSOS+ to maintain it in various system-wide queues and structures. For example, a TCB might be in one of several queues -- the ready queue, a message queue wait queue, a semaphore wait queue, or a memory region wait queue. It might additionally be in a timeout queue.

At pSOS+ startup, a fixed number of TCBs is allocated reflecting the maximum number of concurrently active tasks specified in the pSOS+ Configuration Table entry **kc_ntask**. A TCB is allocated to each task when it is created, and is reclaimed for reuse when the task is deleted. Memory considerations for TCBs are given in the *pSOSystem Programmer's Reference* manual.

A task's **Tid** contains, among other things, the encoded address of the task's TCB. Thus, for system calls that supply **Tid** as input, pSOS+ can quickly locate the target task's TCB. By convention, a **Tid** value of 0 is an alias for the running task. Thus, if 0 is used as the **Tid** in a system call, the target will be the calling task's TCB.

2.5.3 Task Mode Word

Each task carries a mode word that can be used to modify scheduling decisions or control its execution environment:

- Preemption Enabled/Disabled -- If a task has preemption disabled, then so long as it is ready, pSOS+ will continue to run it, even if there are higher priority tasks also ready.
- Roundrobin Enabled/Disabled -- Its effects are discussed in section 2.4.5, "Roundrobin by Timeslicing."
- ASR Enabled/Disabled -- Each task can have an Asynchronous Signal Service Routine (ASR), which must be established by the **as_catch** system call. Asynchronous signals behave much like software interrupts. If a task's ASR is enabled, then an **as_send** system call directed at the task will force it to leave its expected execution path, execute the ASR, and then return to the expected execution path. See section 2.11.1, "The ASR," for more details on ASRs.

- Interrupt Mask Level -- A task can execute optionally at a non-zero interrupt mask level, although this is advisable only for short instruction paths.

A task's mode word is set up initially by the **t_start** call, and can be changed dynamically using the **t_mode** call.

NOTE: To ensure correct operation of your application, you should avoid direct modification of the interrupt mask level in the CPU status register. Use t_mode for such purposes, so that pSOS+ is correctly informed of such changes.

2.5.4 Task Stacks

pSOS+ switches stacks whenever it switches from one running task to another. A task must have at least a supervisor stack. A task must also have a user stack, if it (ASR included) ever executes in the user state; otherwise it will crash. The CPU automatically selects and uses the right stack.

For more information on stack usage, refer to the *pSOSystem Programmer's Reference* manual.

2.5.5 Task Memory

pSOS+ allocates and maintains a task's stacks, but it has no explicit knowledge of a task's code or data areas.

For most applications, application code is memory resident prior to system startup, being either ROM resident or bootloaded. For some systems, a task can be brought into memory just before it is created or started; in which case, memory allocation and/or location sensitivity should be considered.

2.5.6 Death of a Task

A task can terminate itself, or another task. The **t_delete** pSOS+ Service removes a created task by reclaiming its TCB and returning the stack memory segment to Region 0. The TCB is marked as free, and can be reused by a new task.

The proper reclamation of resources such as segments, buffers, or semaphores should be an important part of task deletion. This is particularly true for dynamic applications, wherein parts of the system can be shutdown and/or regenerated on demand.

In general, **t_delete** should only be used to perform self-deletion. The reason is simple. When used to forcibly delete another task, **t_delete** denies that task a chance to perform any necessary cleanup work. A preferable method is to use the **t_restart** call, which forces a task back to its initial entry point. Since **t_restart** can pass an optional argument list, the target task can use this to distinguish between a **t_start**, a meaningful **t_restart**, or a request for self-deletion. In the latter case, the task can return any allocated resources, execute any necessary cleanup code, and then gracefully call **t_delete** to delete itself.

A deleted task ceases to exist insofar as pSOS+ is concerned, and any references to it, whether by name or by **Tid**, will evoke an error return.

2.5.7 Notepad Registers

Each task has 16 software notepad 32-bit registers. They are carried in a task's TCB, and can be set and read using the **t_setreg** and **t_getreg** calls, respectively. The purpose of these registers is to provide to each task, in a standard system-wide manner, a set of named variables that can be set and read by other tasks, including by remote tasks on other processor nodes.

Eight of these notepad registers are reserved for system use. The remaining eight can be used for any application specific purposes.

2.5.8 The Idle Task

At startup, pSOS+ automatically creates and starts an idle task, named **IDLE**, whose sole purpose in life is to soak up CPU time when no other task can run. This task, which runs at priority 0, executes only one instruction -- **STOP**.

Nevertheless, **IDLE** is an important task. It must not be tampered with via **t_delete**, **t_suspend**, **t_setpri**, or **t_mode**, unless you have provided an equivalent task to fulfill this necessary idling function. In certain cases, **IDLE** must be replaced. One such instance is if your hardware or test setup cannot properly support the **STOP** instruction.

IDLE runs in supervisor mode with a supervisor stack allocated from Region 0 with a size equal to **kc_rootsstk**.

2.6 Storage Allocation

pSOS+ storage management services provide dynamic allocation of both variable size segments and fixed size buffers. The system calls are

rn_create	Create a memory region.
rn_ident	Get the ID of a memory region.
rn_delete	Delete a memory region.
rn_getseg	Allocate a segment from a region.
rn_retseg	Return a segment to a region.
pt_create	Create a partition of buffers.
pt_ident	Get the ID of a partition.
pt_delete	Delete a partition of buffers.
pt_getbuf	Get a buffer from a partition.
pt_retbuf	Return a buffer to a partition.

2.6.1 Regions and Segments

A memory region is a user-defined, physically contiguous block of memory. Regions can possess distinctive implicit attributes. For example, one can reside in strictly local RAM, another in system-wide accessible RAM. Regions must be mutually disjoint and can otherwise be positioned on any long word boundary.

Like tasks, regions are dynamic abstract objects managed by pSOS+. A region is created using the **rn_create** call with the following inputs -- its user-assigned name, starting address and length, and **unit_size**. The pSOS+ system call **rn_create** returns a region ID (**RNid**) to the caller. For any other task that knows a region only by name, the **rn_ident** call can be used to obtain a named region's **RNid**.

A segment is a variable-sized piece of memory from a memory region, allocated by pSOS+ on the **rn_getseg** system call. Inputs to **rn_getseg** include a region ID, a segment size that might be anything, and an option to wait until there is sufficient free memory in the region. The **rn_retseg** call reclaims an allocated segment and returns it to a region.

A region can be deleted, although this is rarely used in a typical application. For one thing, deletion must be carefully considered, and is allowed by pSOS+ only if there are no outstanding segments allocated from it, or if the delete override bit was set when the region was created.

2.6.2 Special Region 0

pSOS+ requires at least one region in order to function. The start address and length of this special region, with the name **RN#0** and **Region_id 0**, are specified in the pSOS+ Configuration Table. During pSOS+ startup, pSOS+ first carves a pSOS+ Data Segment from the beginning of Region 0 for its own data area and control structures such as TCBs, etc. A formula to calculate the exact size of this pSOS+ Data Segment is given in the *pSOSystem Programmer's Reference* manual. The remaining block of Region 0 is used for task stacks, as well as any user **rn_getseg** calls.

pSOS+ pre-allocates memory for its own use. That is, after startup, pSOS+ makes no dynamic demands for memory. However, when the **t_create** system call is used to create a new task, pSOS+ will internally generate an **rn_getseg** call to obtain a segment from Region 0 to use as the task's supervisor and user(optional) stacks. Similarly, when **q_vcreate** is used to create a variable length message queue, pSOS+ allocates a segment from Region 0 to store messages pending at the queue.

Note that pSOS+ keeps track of each task's stack segment and each variable length message queue's message storage segment. When a task or variable length queue is deleted, pSOS+ automatically reclaims the segment and returns it to Region 0.

Like any memory region, your application can make **rn_getseg** and **rn_retseg** system calls to Region 0 to dynamically allocate and return variable-sized memory segments. Region 0, by default, queues any tasks waiting there for segment allocation by FIFO order.

2.6.3 Allocation Algorithm

pSOS+ takes a piece at the beginning of the input memory area to use as the region's control block (RNCB). The size of the RNCB varies, depending on the region size and its **unit_size** parameter, described below. A formula giving the size of an RNCB is given in the *pSOSystem Programmer's Reference* manual.

Each memory region has a **unit_size** parameter, specified as an input to **rn_create**. This region-specific parameter is the region's smallest unit of allocation. This unit must be a power of 2, but greater than or equal to 16, bytes. Any segment allocated by **rn_getseg** is always a size equal to the nearest multiple of **unit_size**. For example, if a region's **unit_size** is

32 bytes, and an **rn_getseg** call requests 130 bytes, then a segment with 5 units or 160 bytes will be allocated. A region's length cannot be greater than 32,767 times the **unit_size** of the region.

The **unit_size** specification has a significant impact on (1) the efficiency of the allocation algorithm, and (2) the size of the region's RNCB. The larger the **unit_size**, the faster the **rn_getseg** and **rn_retseg** execution, and the smaller the RNCB.

The pSOS+ region manager uses an efficient heap management algorithm. A region's RNCB holds an allocation map and an heap structure used to manage an ordered list of free segments. By maintaining free segments in order of decreasing size, an **rn_getseg** call only needs to check the first such segment. If the segment is too small, then allocation is clearly impossible. The caller can wait, wait with timeout, or return immediately with an error code. If the segment is large enough, then it will be split. One part is returned to the calling task. The other part is re-entered into the heap structure.

When **rn_retseg** returns a segment, pSOS+ always tries to merge it with its neighbor segments, if one or both of them happen to be free. Merging is fast, since the neighbor segments can be located without searching. The resulting segment is then re-entered into the heap structure.

2.6.4 Partitions and Buffers

A memory partition is a user-defined, physically contiguous block of memory, divided into a set of equal-sized buffers. Aside from having different buffer sizes, partitions can possess distinctive implicit attributes. For example, one can reside in strictly local RAM, another in system-wide accessible RAM. Partitions must be mutually disjoint.

Like regions, partitions are dynamic abstract objects managed by pSOS+. A partition is created using the **pt_create** call with the following inputs -- its user-assigned name, starting address and length, and **buffer_size**. The system call **pt_create** returns a partition ID (**PTid**) assigned by pSOS+ to the caller. For any other task that knows a partition only by name, the **pt_ident** call can be used to obtain a named partition's **PTid**.

pSOS+ takes a small piece at the beginning of the input memory area to use as the partition's control block (PTCB). The rest of the partition is organized simply as a pool of equal-sized buffers. Because of this simple organization, the **pt_getbuf** and **pt_retbuf** system calls are highly efficient.

A partition has the following limits -- it must start on a long-word boundary and its buffer size must be a power of 2, but greater than or equal to 4 bytes.

Partitions can be deleted, although this is rarely used in a typical application. For one thing, deletion must be carefully considered, and is allowed by pSOS+ only if there are no outstanding buffers allocated from it.

Partitions can be used, in a tightly-coupled multiprocessor configuration, for efficient data exchange between processor nodes. For a complete discussion of shared partitions, see Chapter 3, "pSOS+m Multiprocessing Kernel."

2.7 Communication, Synchronization, Mutual Exclusion

A pSOS+ based application is generally partitioned into a set of tasks and interrupt service routines (ISRs). Conceptually, each task is a thread of independent actions that can execute concurrently with other tasks. However, cooperating tasks need to exchange data, synchronize actions, or share exclusive resources. To service task-to-task as well as ISR-to-task communication, synchronization, and mutual exclusion, pSOS+ provides three sets of facilities -- message queues, events, and semaphores.

2.8 The Message Queue

Message queues provide a highly flexible, general-purpose mechanism to implement communication and synchronization. The related system calls are listed below:

q_create	Create a message queue.
q_ident	Get the ID of a message queue.
q_delete	Delete a message queue.
q_receive	Get / wait for message from a queue.
q_send	Post a message at end of a queue.
q_urgent	Put a message at head of a queue.
q_broadcast	Broadcast a message to a queue.

Like a task, a message queue is an abstract object, created dynamically using the **q_create** system call. **q_create** accepts as input a user-assigned name and several characteristics, including whether tasks waiting for messages there will wait first-in-first-out, or by task priority, whether the message queue has a limited length, and whether a set of message buffers will be reserved for its private use.

A queue is not explicitly bound to any task. Logically, one or more tasks can send messages to a queue, and one or more tasks can request messages from it. A message queue therefore serves as a many-to-many communication switching station.

Consider this simple many-to-1 communication example. A server task can use a message queue as its input request queue. Several client tasks independently send request messages to this queue. The server task waits at this queue for input requests, processes them, and goes back for more -- a simple single queue, single server implementation.

The number of message queues in your system is limited only by the **kc_nqueue** specification in the pSOS+ Configuration Table.

A message queue can be deleted using the **q_delete** system call. If one or more tasks are waiting there, they will be removed from the wait queue and returned to the ready state. When they run, each task will have returned from their respective **q_receive** call with an error code (Queue Deleted). On the other hand, if there are messages posted at the queue, then pSOS+ will reclaim the message buffers and all message contents are lost. Message buffers are covered in section 2.8.3, "Messages and Message Buffers."

2.8.1 The Queue Control Block

Like a **Tid**, a message queue's **Qid** carries the location of the queue's control block (QCB), even in a multiprocessor configuration. This is an important notion, since using the **Qid** to reference a message queue totally eliminates the need to search for its control structure.

A QCB is allocated to a message queue when it is created, and reclaimed for re-use when it is deleted. This structure contains the queue's name and ID, wait-queueing method, and message queue length and limit. Memory considerations for QCBs are given in the *pSOSystem Programmer's Reference* manual.

2.8.2 Queue Operations

A queue usually has two types of users- sources and sinks. A source posts messages, and can be a task or an ISR. A sink consumes messages, and can be another task or (with certain restrictions) an ISR.

There are three different ways to post a message -- **q_send**, **q_urgent**, and **q_broadcast**.

When a message arrives at a queue, and there is no task waiting, it is copied into a message buffer taken from either the shared or (if it has one) the queue's private, free buffer pool. The message buffer is then entered into the message queue. A **q_send** call puts a message at the end of the message queue. **q_urgent** inserts a message at the front of the message queue.

When a message arrives at a queue, and there are one or more tasks already waiting there, then the message will be given to the first task in the wait queue. No message buffer will be used. That task then leaves the queue, and becomes ready to run.

The **q_broadcast** system call broadcasts a message to all tasks waiting at a queue. This provides an efficient method to wake up multiple tasks with a single system call.

There is only one way to request a message from a queue -- the **q_receive** system call. If no message is pending, the task can elect to wait, wait with timeout, or return unconditionally. If a task elects to wait, it will either be by first-in-first-out or by task priority order, depending on the specifications given when the queue was created. If the message queue is non-empty, then the first message in the queue will be returned to the caller. The message buffer that held that message is then released back to the shared or the queue's private free buffer pool.

2.8.3 Messages and Message Buffers

Messages are fixed length, consisting of 4 long-words. A message's content is entirely dependent on the application. It can be used to carry data, pointer to data, data size, the sender's **Tid**, a response queue **Qid**, or some combination of the above. In the degenerate case where a message is used purely for synchronization, it might carry no information at all.

When a message arrives at a message queue and no task is waiting, the message must be copied into a message buffer that is then entered into the message queue.

A message buffer is 6 long words. The extra two long-words are used as a link field and a message length field by pSOS+. At startup, pSOS+ allocates a shared pool of free message buffers; the size of this pool is equal to the **kc_nmsgbuf** entry in the pSOS+ Configuration Table.

A message queue can be created to use either a pool of buffers shared among many queues or its own private pool of buffers. In the first case, messages arriving at the queue will use free buffers from the shared pool on an as-needed basis. In the second case, a number of free buffers equal to the queue's maximum length will be taken from Region 0 and set aside for the private use of the message queue.

2.8.4 Two Examples of Queue Usage

The examples cited below, and depicted in Figure 5, illustrates the ways in which the generalized message queue facility can be used to implement various synchronization requirements.

The first example typifies the straightforward use of a message queue as a FIFO queue between one or more message sources, and one or more message sinks. Synchronization provided by a single queue is one-way and non-interlocked. That is, a message sink synchronizes its activities to the arrival of a message to the queue, but a message source does not synchronize to any queue or sink condition -- it can elect to produce messages at its own pace.

The second example utilizes two queues to close the synchronization loop, and provide interlocked communication. A task that is a message sink to one queue is a message source to the other, and vice-versa. Task A sends a message to queue X, and does not continue until it receives a message from queue Y. Task B synchronizes itself to the arrival of a message to queue X, and responds by sending an acknowledgment message to queue Y. The result is that tasks A and B interact in an interlocked, coroutine-like fashion.

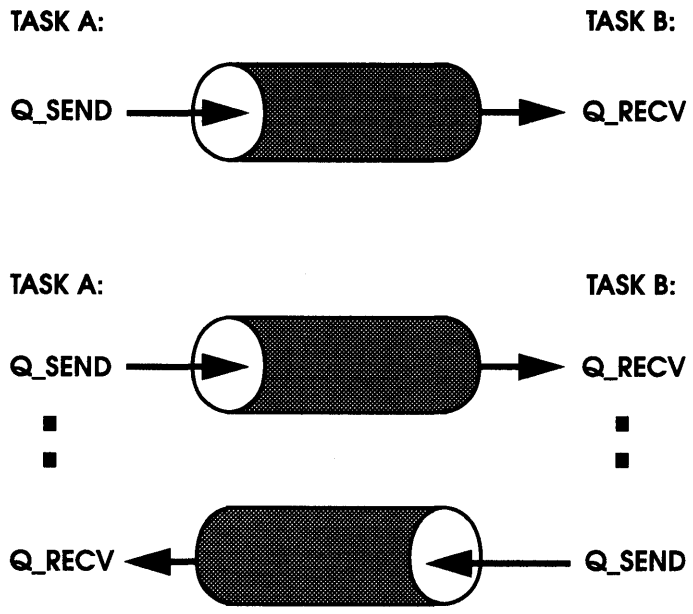

Figure 5. One Way and Two Way Queue Synchronization

2.8.5 Variable Length Message Queues

Recall that ordinary message queues use fixed-length 16-byte messages. While 16 bytes is adequate for most purposes, in some cases it is convenient to use messages of differing sizes, particularly larger messages. pSOS+ supports a special type of message queue called a *variable length message queue.* A variable length message queue can accept messages of any length up to a maximum specified when the queue is created.

Internally pSOS+ implements variable length message queues as a special type of ordinary queue. That is, ordinary and variable length message queues are not different objects, but rather, different forms of the same object.

Although they are implemented using the same underlying object, pSOS+ provides a complete family of services to create, manage, and use variable length message queues. These services are as follows:

q_vcreate	Create a variable length message queue
q_vident	Get the ID of a variable length message queue
q_vdelete	Delete a variable length message queue
q_vreceive	Get or wait for message from a variable length message queue
q_vsend	Post a message at end of a variable length message queue
q_vurgent	Put a message at head of a variable length message queue
q_vbroadcast	Broadcast a message to a variable length message queue

A variable length queue is created with the **q_vcreate** service call. In addition to **name** and **flags** the caller provides two additional input parameters. The first specifies the queue's *maximum message length*. A message of any length up to this maximum can be sent to the queue. Any attempt to send a message larger than a queue's maximum message length results in an error. The second parameter specifies the queue's *maximum message queue length*. This is the maximum number of messages that can be waiting at the queue simultaneously.

Unlike ordinary queues, which use buffers from the system-wide buffer pool for message storage, variable length queues always store messages in buffers that are allocated from region 0 when the queue is created. These buffers are then available for the exclusive use of the queue. They are never shared with other queues and they are only returned to region 0 if and when the queue is deleted. Note, this is the only time when storage for a pSOS+ object is allocated from region 0.

Once a variable length message queue has been created, variable length message are sent and received using the **q_vsend**, **q_vurgent**, **q_vbroadcast**, and **q_vreceive** service calls. The calls operate exactly like their ordinary counterparts (**q_send**, **q_urgent**, **q_broadcast**, and **q_vreceive**), except the caller must provide an additional parameter that specifies the length of the message.

The remaining two variable length message queue services, **q_vident** and **q_vdelete** are identical to their ordinary counterparts (**q_ident** and **q_delete**) in every respect.

Note that although ordinary and variable length message queues are implemented using the same underlying object, service calls cannot be mixed. For example, **q_send** cannot be used to post a message to a variable length message queue. Similarly, **q_vsend** cannot be used to send a message to an ordinary queue. There is one exception -- **q_ident** and **q_vident** are identical. When searching for the named queue, both return the first queue encountered that has the specified name, regardless of the queue type.

2.9 Events

pSOS+ provides a set of synchronization-by-event facilities. Each task has thirty-two events flags it can wait on, bit-wise encoded in a 32-bit word. The high 16 bits are reserved for system use. The lower 16 event flags are completely user definable.

Two pSOS+ system calls provide synchronization by events between tasks and between tasks and ISRs:

ev_receive	Get or wait for events.
ev_send	Send events to a task.

ev_send is used to send one or more events to another task. With **ev_receive**, a task can wait for, with or without timeout, or request without waiting, one or more of its own events. One important feature of events is that a task can wait for one event, one of several events (or), or all of several events (and).

2.9.1 Event Operations

Events are independent of each other. The **ev_receive** call permits synchronization to the arrival of one or more events, qualified by an AND or OR condition. If all the required event bits are on (i.e. pending), then the **ev_receive** call resets them and returns immediately. Otherwise, the task can elect to return immediately or block until the desired event(s) have been received.

A task or ISR can send one or more events to another task. If the target task is not waiting for any event, or if it is waiting for events other than those being sent, **ev_send** simply turns the event bit(s) on, which makes the events pending. If the target task is waiting for some or all of the events being sent, then those arriving events that match are used to satisfy the waiting task. The other non-matching events are made pending, as before. If the requisite event condition is now completely satisfied, the task is unblocked and made ready-to-run; otherwise, the wait continues for the remaining events.

2.9.2 Events Versus Messages

Events differ from messages in the following sense:

- An event can be used to synchronize with a task, but it cannot directly carry any information.
- Topologically, events are sent point to point. That is, they explicitly identify the receiving task. A message, on the other hand, is sent to a message queue. In a multireceiver case, a message sender does not necessarily know which task will receive the message.
- One **ev_receive** call can condition the caller to wait for multiple events. **q_receive**, on the other hand, can only wait for one message from one queue.
- Messages are automatically buffered and queued. Events are neither counted nor queued. If an event is already pending when a second, identical one is sent to the same task, the second event will have no effect.

2.10 Semaphores

pSOS+ provides a set of familiar semaphore operations. In general, they are most useful as resource tokens in implementing mutual exclusion. The related system calls are listed below:

sm_create	Create a semaphore.
sm_ident	Get the ID of a semaphore.
sm_delete	Delete a semaphore.
sm_p	Get / wait for a semaphore token.
sm_v	Return a semaphore token.

Like a message queue, a semaphore is an abstract object, created dynamically using the **sm_create** system call. **sm_create** accepts as input a user-assigned name, an initial count, several characteristics, including whether tasks waiting for the semaphore will wait first-in-first-out, or by task priority. The initial count parameter should reflect the number of available "tokens" at the semaphore. **sm_create** assigns a unique ID, the **SMid**, to each semaphore.

The number of semaphores in your system is limited only by the **kc_nsema4** specification in the pSOS+ Configuration Table.

A semaphore can be deleted using the **sm_delete** system call. If one or more tasks are waiting there, they will be removed from the wait queue and returned to the ready state. When they run, each task will have returned from their respective **sm_p** call with an error code (Semaphore Deleted). Any attempt to return tokens to it (**sm_v** calls) will be rejected.

2.10.1 The Semaphore Control Block

Like a **Qid**, a semaphore's **SMid** carries the location of the semaphore control block (SMCB), even in a multiprocessor configuration. This is an important notion, since using the **SMid** to reference a semaphore eliminates completely the need to search for its control structure.

A SMCB is allocated to a semaphore when it is created, and reclaimed for re-use when it is deleted. This structure contains the semaphore's name and ID, the token count, and wait-queueing method. It also contains the head and tail of a doubly linked task wait queue.

Memory considerations for SMCBs are given in the *pSOSystem Programmer's Reference* manual.

2.10.2 Semaphore Operations

pSOS+ supports the two traditional P and V semaphore primitives. The **sm_p** call requests a token. If the semaphore token count is non-zero, then **sm_p** decrements the count and the operation is successful. If the count is zero, then the caller can elect to wait, wait with timeout, or return unconditionally. If a task elects to wait, it will either be by first-in-first-out or by task priority order, depending on the specifications given when the semaphore was created.

The **sm_v** call returns a semaphore token. If no tasks are waiting at the semaphore, then **sm_v** simply increments the semaphore token count. If tasks are waiting, then the first task in the semaphore's wait list is released from the list and made ready to run.

2.11 Asynchronous Signals

Each task can optionally have an Asynchronous Signal Service Routine (ASR). The ASR's purpose is to allow a task to have two asynchronous parts -- a main body and an ASR. In essence, just as one task can execute asynchronously from another task, an ASR provides a similar capability within a task.

Using signals, one task or ISR can selectively force another task out of its normal locus of execution -- that is, from the task's main body into its ASR. Signals provide a "software interrupt" mechanism. This asynchronous communications capability is invaluable to many system designs. Without it, workarounds must depend on synchronous services such as messages or events, which, even if possible, suffer a great loss in efficiency.

There are three related system calls:

as_catch	Establish a task's ASR.
as_send	Send signals to a task.
as_return	Return from an ASR.

An asynchronous signal is a user-defined condition. Each task has 32 signals, encoded bit-wise in a long word. To receive signals, a task must establish an ASR using the **as_catch** call. The **as_send** call can be used to send one or more asynchronous signals to a task, thereby forcing the task, the next time it is dispatched, to first go to its ASR. At the end of an ASR, a call to **as_return** allows pSOS+ to return the task to its original point of execution.

2.11.1 The ASR

A task can have only one active ASR, established using the **as_catch** call. A task's ASR executes in the task's context -- from the outside, it is not possible to discern whether a task is executing in its main code body or its ASR.

The **as_catch** call supplies both the ASR's starting address and its initial mode of execution. This mode replaces the mode of the task's main code body (see section 2.5.3, "Task Mode Word") as long as the ASR is executing. It is used to control the ASR's execution behavior, including whether it is preemptible, whether it executes in the user or supervisor state, and whether or not further asynchronous signals are accepted. Typically, ASRs execute with asynchronous signals disabled. Otherwise, the ASR must be programmed to handle re-entrancy.

The details of how an ASR gains control is processor-specific; this information can be found in the *pSOSystem Programmer's Reference* manual.

A task can disable and enable its ASR selectively by calling **t_mode**. Any signals received while a task's ASR is disabled are left pending. When re-enabled, an ASR will receive control if there are any pending signals.

2.11.2 Asynchronous Signal Operations

The **as_send** call simply makes the specified signals pending at the target task, without affecting its state or when it will run. If the target task is not the running task, its ASR takes over only when it is next dispatched to run. If the target is the running task, which is possible only if the signals are sent by the task itself or, more likely, by an ISR, then the running task's course changes immediately to the ASR.

2.11.3 Signals Versus Events

Despite their resemblance, asynchronous signals are fundamentally different from events, as follows:

- To synchronize to an event, a task must explicitly call **ev_receive**. **ev_send** by itself has no effect on the receiving task's state. By contrast, **as_send** can unilaterally force the receiving task to execute its ASR.
- From the perspective of the receiving task, response to events is synchronous; it occurs only after a successful **ev_receive** call. Response to signals is asynchronous; it can happen at any point in the task's execution. Note that, while this involuntary-response behavior is by design, it can be modified to some extent by using **t_mode** to disable (i.e. postpone) asynchronous signal processing.

2.12 Time Management

Time management provides the following functions:

- Maintain calendar time and date.
- Timeout (optional) a task that is waiting for messages, semaphores, events or segments.
- Wake up or send an alarm to a task after a designated interval or at an appointed times.
- Track the running task's timeslice, and mechanize roundrobin scheduling.

These functions depend on periodic timer interrupts, and will not work in the absence of a real-time clock or timer hardware.

The explicit time management system calls are:

tm_tick	Inform pSOS+ of clock tick arrival.
tm_set	Set time and date.
tm_get	Get time and date.
tm_wkafter	Wakeup task after interval.
tm_wkwhen	Wakeup task at appointed time.
tm_evafter	Send event to task after interval.
tm_evwhen	Send event to task at appointed time.
tm_cancel	Cancel an alarm timer.

2.12.1 The Time Unit

The system time unit is a clock tick, defined as the interval between **tm_tick** system calls. This pSOS+ call is used to announce to pSOS+ the arrival of a clock tick -- it is normally called from the real-time clock ISR, on each timer interrupt. The frequency of **tm_tick** determines the granularity of the system time-base. Obviously, the higher the frequency, the higher the time resolution for timeouts, etc. On the other hand, processing each clock tick takes a small amount of system overhead.

You can specify this clock tick frequency in the pSOS+ Configuration Table as **kc_ticks2sec**. For example, if this value is specified as 100, the system time manager will interpret 100 **tm_tick** system calls to be one second, real-time.

2.12.2 Time and Date

pSOS+ maintains true calendar time and date, including perpetual leap year compensation. A pair of simple pSOS+ system calls, **tm_set** and **tm_get**, allows you to set and obtain the date and time of day. Time resolution is accurate to system time ticks.

No elapsed tick counter is included, since this can be easily maintained by your own code. For example, your real-time clock ISR can, in addition to calling **tm_tick** on each clock interrupt, increment a 32-bit global counter variable.

2.12.3 Timeouts

Implicitly, pSOS+ uses the time manager to provide a timeout facility to other system calls, e.g. **q_receive**, **ev_receive**, **sm_p**, and **rn_getseg**.

pSOS+ uses a proprietary timing structure and algorithm, which, in addition to being efficient, guarantees constant-time operations. Both task entry into and removal from the timeout state are performed in constant time -- no search loops are required.

If a task is waiting, say for message (**q_receive**), with timeout, and the message arrives in time, then the task is simply removed from the timing structure, given the message, and made ready to run. If the message does not arrive before the time interval expires, then the task will be given an error code indicating timeout, and made ready to run.

Timeout is measured in ticks. If **kc_ticks2sec** is 100, and an interval of 50 milliseconds is required, then a value of 5 should be specified. Timeout intervals are 32 bits wide, allowing a maximum of 2^{32} ticks. A timeout value of n will expire on the nth forthcoming tick. Since the system call can come anywhere between two ticks, this implies that the real-time interval will be between n-1 and n ticks.

2.12.4 Absolute Versus Relative Timing

There are two ways a task can specify timing -- relative or absolute. Relative timing is specified as an interval, measured in ticks. Absolute timing is specified as an appointed calendar date and time. The system calls **tm_wkafter** and **tm_evafter** accept relative timing specifications. The system calls **tm_wkwhen** and **tm_evwhen** accept absolute time specifications.

Note that absolute timing is affected by any **tm_set** calls that change the calendar date and time, whereas relative timings are not affected. In addition, use of absolute time specifications might require additional time manipulations.

2.12.5 Wakeups Versus Alarms

There are two distinct ways a task can respond to timing. The first way is to go to sleep (i.e. block), and wake up at the desired time. This synchronous method is supported by the **tm_wkafter** and **tm_wkwhen** calls. The second way is to set an alarm timer, and then continue running. This asynchronous method is supported by **tm_evafter** and **tm_evwhen**. When the alarm timer goes off, pSOS+ will internally call **ev_send** to send the designated events to the task. Of course, the task must call **ev_receive** in order to test or wait for the scheduled event.

Alarm timers offer several interesting features. First, the calling task can execute while the timer is counting down. Second, a task can arm more than one alarm timer, each set to go off at different times, corresponding to multiple expected conditions. This multiple alarm capability is especially useful in implementing nested timers, a common requirement in more sophisticated communications systems. Third, alarm timers can be canceled using the **tm_cancel** call.

In essence, the wakeup mechanism is useful only in timing an entire task. The alarm mechanism can be used to time transactions within a task.

2.12.6 Timeslice

If the running task's mode word (see section 2.5.3, "Task Mode Word") has its roundrobin bit and preemptible bit on, then pSOS+ will countdown the task's assigned timeslice. If it is still running when its timeslice is down to zero, then roundrobin scheduling will take place. Details of the roundrobin scheduling can be found in section 2.4.5, "Roundrobin by Timeslicing."

You can specify the amount of time that constitutes a full timeslice in the pSOS+ Configuration Table as **kc_ticks2slice**. For instance, if that value is 10, and the **kc_ticks2sec** is 100, then a full timeslice is equivalent to about one-tenth of a second. The countdown or consumption of a timeslice is somewhat heuristic in nature, and might not exactly reflect the actual elapsed time a task has been running.

2.13 Interrupt Service Routines

Interrupt service routines (ISRs) are critical to any real-time system. On one side, an ISR handles interrupts, and performs whatever minimum action is required, to reset a device, to read/write some data, etc. On the other side, an ISR might drive one or more tasks, and cause them to respond to, and process, the conditions related to the interrupt.

An ISR's operation should be kept as brief as possible, in order to minimize masking of other interrupts at the same or lower levels. Normally, it simply clears the interrupt condition and performs the necessary physical data transfer. Any additional handling of the data should be deferred to an associated task with the appropriate (software) priority. This task can synchronize its actions to the occurrence of a hardware interrupt, by using either a message queue, events flag, semaphores, or ASR.

2.13.1 Interrupt Entry

Interrupts should be directly vectored to the user-supplied ISRs. pSOS+ does not interfere with interrupt entry, in order to achieve the fastest response possible.

2.13.2 Synchronizing With Tasks

An ISR usually communicates with one or more tasks, either directly, or indirectly as part of its input/output transactions. The nature of this communication is usually to drive a task, forcing it to run and handle the interrupting condition. This is similar to the task-to-task type of communication or synchronization, with two important differences.

First, an ISR is usually a communication/synchronization source -- it often needs to return a semaphore, or send a message or an event to a task. An ISR is rarely a communication sink -- it cannot wait for a message or an event.

Second, a system call made from an ISR will always return immediately to the ISR, without going through the normal pSOS+ dispatch. For example, even if an ISR sends a message and wakes up a high priority task, pSOS+ must nevertheless return first to the ISR. This deferred dispatching is necessary, since the ISR must be allowed to complete.

pSOS+ allows an ISR to make any of the synchronization sourcing system calls, including **q_send**, **q_urgent** and **q_broadcast** to post messages to message queues, **sm_v** to return a semaphore, and **ev_send** to send events to tasks.

A typical system implementation, for example, can use a message queue for this ISR-to-task communication. A task requests and waits for a message at the queue. An ISR sends a message to the queue, thereby unblocking the task and making it ready to run. The ISR then exits the interrupt level using the **I_RETURN** entry into pSOS+. Among other things, **I_RETURN** causes pSOS+ to dispatch to run the highest priority task, which can be the interrupted running task, or the task just awakened by the ISR. The message, as usual, can be used to carry data, pointers to data, or simply for synchronization.

In some applications, an ISR might additionally have the need to dequeue messages from a message queue. For example, a message queue might be used to hold a chain of commands. Tasks needing service will send command messages to the queue. When an ISR finishes one command, it checks to see if the command chain is now empty. If not, then it will dequeue the next command in the chain and start it. To support this type of implementation, pSOS+ allows an ISR to make **q_receive** system calls to obtain messages from a queue, and **sm_p** calls to acquire a semaphore. Note, however, that these calls must use the no-wait option, so that the call will return whether or not a message or semaphore is available.

2.13.3 System Calls Allowed From an ISR

The restricted subset of pSOS+ system calls that can be issued from an ISR are as follows:

as_send	Send asynchronous signals to a task (local task).
ev_send	Send events to a task (local task).
k_fatal	Abort and enter fatal error handler.
k_terminate	Terminate a failed node (pSOS+m only).
pt_getbuf	Get a buffer from a partition (local partition).
pt_retbuf	Return a buffer to a partition (local partition).

q_broadcast	Broadcast a message to an ordinary queue (local queue).
q_receive	Get a message from an ordinary message queue (no-wait and local queue).
q_send	Post message to end of an ordinary message queue (local queue).
q_urgent	Post message at head of an ordinary message queue (local queue).
q_vbroadcast	Broadcast a variable length message to queue (local queue).
q_vreceive	Get a message from a variable length message queue (no-wait and local queue).
q_vsend	Post message to end of a variable length message queue (local queue).
q_vurgent	Post message at head of a variable length message queue (local queue).
sm_p	Acquire a semaphore (no-wait and local semaphore).
sm_v	Return a semaphore (local semaphore).
t_getreg	Get a task's software register (local task).
t_resume	Resume a suspended task (local task).
t_setreg	Set a task's software register (local task).
tm_get	Get time and date.
tm_set	Set time and date.
tm_tick	Announce a clock tick to pSOS+.

As noted earlier, since an ISR cannot block, a **q_receive**, **q_vreceive**, or **sm_p** call from an ISR must use the no-wait, i.e. unconditional return, option. Also, since remote service calls block, the above services can only be called from an ISR if the referenced object is local.

All other pSOS+ system calls are either not meaningful in the context of an ISR, or can be functionally served by another system call. Making calls not listed above from an ISR will lead to dangerous race conditions, and unpredictable results.

2.14 Fatal Errors and the Shutdown Procedure

Most error conditions resulting from system calls, for example parametric and temporary resource exhaustion errors, are non-fatal. These are simply reported back to the caller. A few error conditions prevent continued operation. This class of errors, known as *fatal errors*, include startup configuration defects, internal resource exhaustion conditions, and various other non-recoverable conditions. In addition, your application software can, at any time, generate a fatal error by making the system call **k_fatal**.

Every fatal error has an associated failure code that defines the cause of the fatal error. The *pSOSystem Programmer's Reference* lists failure codes that are generated by pSOSystem. Note that all pSOSystem-generated failure codes are of the form 0xXFXX where X is a hexadecimal digit (that is, the third nibble is always F and the code never exceeds 0xFFFF). Failure codes equal to or greater than 0x10000 are reserved for use by application code. In this case, the failure code is provided as an input parameter to **k_fatal** or **k_terminate** (in multiprocessor systems).

When a fatal error occurs, whether generated internally by pSOSystem or by a call to **k_fatal** or **k_terminate**, pSOS+ passes control to an internal *fatal error handler*. In single processor systems, the fatal error handler simply performs the *shutdown procedure* described below. In multiprocessor systems it has the additional responsibility of removing the node from the multiprocessor system (see Chapter 3, "pSOS+m Multiprocessing Kernel").

The shutdown procedure is a procedure whereby pSOS+ attempts to halt execution in the most orderly manner possible. pSOS+ first examines the pSOS+ Configuration Table entry **kc_fatal**. If this entry is non-zero, pSOS+ jumps to this address. If **kc_fatal** is zero, and the pROBE+ System Debug/Analyzer is present, then pSOS+ simply passes control to the System Failure entry of pROBE+. Refer to the *pROBE+ User's Manual* for a description of pROBE+ behavior in this case. Finally, if pROBE+ is absent, pSOS+ internally executes a divide-by-zero to cause a deliberate divide-by-zero exception. This will hopefully pass control to a ROM monitor or other low-level debug tool.

In all cases, pSOS+ makes certain information regarding the nature of the failure available to the entity receiving control. Refer to the *pSOSystem Programmer's Reference* for a detailed description of this information.

2.15 Tasks Using Other Components

Integrated Systems offers many other system components that can be used in systems with the pSOS+ kernel. While these components are easy to install and use, they require special consideration with respect to their internal resources and multitasking.

During normal operation, components internally allocate and hold resources on behalf of calling tasks. Some resources are held only during execution of a service call, while others are held indefinitely on based the state of the task. For example: in pHILE+, control information is kept whenever files are open. The pSOS+ service calls **t_restart** and **t_delete** asynchronously alter the execution path of a task and present special problems relative to management of these resources.

The next subsections discuss deletion and restart-related issues in detail and present recommended methods for performing these operations.

2.15.1 Deleting Tasks That Use Components

To avoid permanent loss of component resources, pSOS+ does not allow deletion of a task that is holding any such resource. Instead, **t_delete** returns an error code, which indicates that the task to be deleted holds one or more resources.

The exact conditions under which components hold resources are complex. In general, any task that has made a component service call might be holding resources. But all components provide a facility for returning all of their task-related resources, via a single service call. We recommend that these calls be made prior to calling **t_delete**.

pHILE+, pNA+ and pREPC+ can hold resources that must be returned before a task can be deleted. These resources are returned by calling **close_f**(0), **close**(0) and **fclose**(0) respectively. Since pREPC+ calls pHILE+, and pHILE+ calls pNA+ (if NFS is in use), these services must be called in the correct order. Below is a sample code fragment that a task can use to delete itself:

```
fclose(0);      /* return pREPC resources */
close_f(0);     /* return pHILE resources */
close(0);       /* return pNA resources */
t_delete(0);    /* and commit suicide */
```

Obviously, calls to components not in use should be omitted.

Since only the task to be deleted can make the necessary close calls, the simplest way to delete a task is to restart the task, passing arguments to it that indicate that the task should delete itself. (Of course, the task code must be written to check its arguments and behave accordingly.)

2.15.2 Restarting Tasks That Use Components

pSOS+ allows a task to be restarted regardless of its current state. Check the manual for each component to determine its behavior under restart.

It is possible to restart a task while the task is executing code within the components themselves. Consider the following example:

1. Task A makes a pHILE+ call.
2. While executing pHILE+ code, task A is preempted by task B.
3. Task B then restarts task A.

In such situations, pHILE+ will correctly return resources as required. However, a file system volume might be left in an inconsistent state. For example, if **t_restart** interrupts a **create_f** operation, a file descriptor (FD) might have been allocated but not the directory entry. As a result, an FD could be permanently lost. But, pHILE+ is aware of this danger, and returns a warning, via the **t_restart**. When such a warning code is received from pHILE+, **verify_vol** should be used to detect and correct any resulting volume inconsistencies.

All components are notified of task restarts, so expect such warnings from any of them.

(Blank Page)

3 pSOS+m Multiprocessing Kernel

pSOS+m is the multiprocessing version of the pSOS+ real-time multitasking operating system kernel. It extends many of the pSOS+ system calls to operate seamlessly across multiple processing *nodes*.

This chapter is designed to supplement the information provided in Chapter 2, "pSOS+ Real-Time Kernel." It covers those areas in which the functionality of pSOS+m differs from that of pSOS+.

3.1 System Overview

The pSOS+m kernel is designed so that tasks that make up an application can reside on several processor nodes and still exchange data, communicate, and synchronize exactly as if they are running on a single processor. To support this, pSOS+m allows systems calls to operate across processor boundaries, system-wide. Processing nodes can be connected via any type of connection; for example, shared memory, message-based buses, or custom links, to name a few.

pSOS+m is designed for *functionally-divided* multiprocessing systems. This is the best model for most real-time applications, given the dedicated nature of such applications and their need for deterministic behavior. Each processor executes and manages a separate, often distinct, set of functions. Typically, the decomposition and assignment of functions is done prior to runtime, and is thus permanent (as opposed to task reassignment or load balancing).

The latest version of pSOS+m incorporates facilities that support the following:

Soft Fail A processing node can suffer a hardware or software failure, and other nodes will continue running.

Hot Swap New nodes can be inserted or removed from a system without shutting down.

3.2 Software Architecture

pSOS+m implements a master - slave architecture. As shown in Figure 6, every pSOS+m system must have exactly one node, called the *master node*, which manages the system and coordinates the activities of all other nodes, called s*lave nodes*. The master node must be present when the system is initialized and must remain in the system at all times. In addition to the master, a system may have anywhere between zero and 16382 slave nodes. Unlike the master node, slave nodes may join, exit, and rejoin the system at any time.

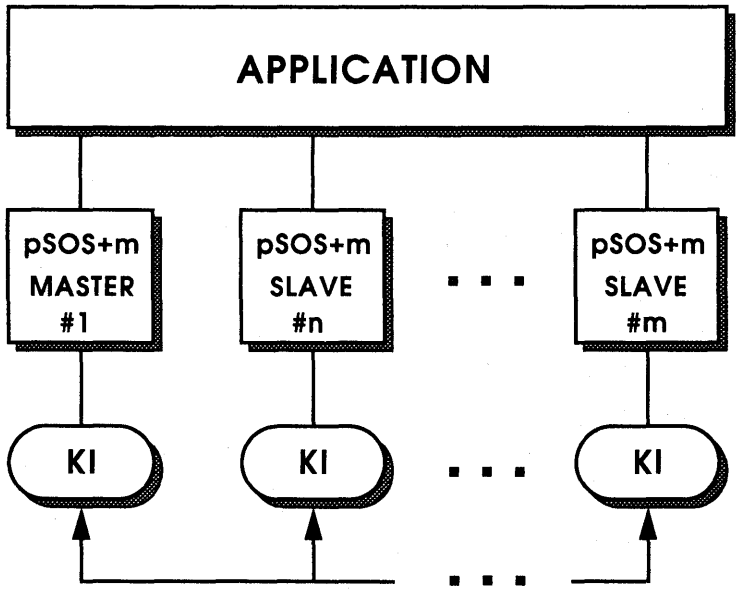

Figure 6. pSOS+m Layered Approach

pSOS+m itself is entirely hardware independent. It makes no assumptions about the physical media connecting the processing nodes, or the topology of the connection. This interconnect medium can be a memory bus, a network, a custom link, or a combination of the above. To perform interprocessor communication, pSOS+m calls a user-provided communication layer called the *Kernel Interface* (KI). The interface between pSOS+m and the KI is standard and independent of the interconnect medium.

In addition to the KI and the standard pSOS+ Configuration table, pSOS+m requires a user-supplied Multiprocessor Configuration Table (MPCT) that defines application-specific parameters.

3.3 Node Numbers

Every node is identified by a user-assigned 32-bit node number. A node number must be unique; that is, no two nodes can have the same number. Node numbers must be greater than or equal to 1 and less than or equal to the maximum node number specified in the Multiprocessor Configuration Table entry **mc_nnode**. Since node numbers must be unique, **mc_nnode** also determines the maximum number of nodes that can be in the system. However, a system may have less than **mc_nnode** nodes if not all node numbers are in use.

Node number 1 designates the master node. All other nodes are slave nodes. One node in your system must be assigned node number 1.

3.4 Objects

pSOS+ is an object-oriented kernel. Object classes include tasks, memory regions, memory partitions, message queues, and semaphores. In a pSOS+m multiprocessor system, the notion of objects transcends node boundaries. Objects (e.g. a task or queue) can be reached or referenced from any node in the system exactly and as easily as if they are all running on a single CPU.

3.4.1 Global Objects

On every object-creation system call, there is a flag parameter, **EXPORT**, which can be used to declare whether the object will be known only locally, or globally to all other nodes in the system. Task, message queue, and semaphore objects can be declared as local or global. Memory partitions can also be declared as global, although this is useful only in a shared memory multiprocessor system where the partition is contained in an area addressable by multiple nodes. Memory region objects can only be local.

An object should be exported only if it will be referenced by a node other than its node of residence, since an exported (i.e. global) object requires management and storage not only on the resident node but also on the master node.

3.4.2 Object ID

Each object, local or global, is known system-wide by two identities -- a user-assigned 32-bit name and a unique pSOS-assigned 32-bit run-time ID. This ID, when used as input on system calls, is used by pSOS+m to locate the object's node of residence as well as its control structure on that node.

This notion of a system-wide object ID is a critical element that enables pSOS+m systems calls to be effective system-wide; that is, transparently across nodes. The application program never needs to possess any explicit knowledge, a priori or acquired, regarding an object's node of residence.

3.4.3 Global Object Tables

Every node running pSOS+ or pSOS+m has a Local Object Table that contains entries for local objects. In a multiprocessor system, every node also has a Global Object Table. A slave node's Global Object Table contains entries for objects that are resident on the slave node and exported for use by other nodes. The master node's global object table contains entries for every exported object in the system, *regardless of its node of residence.*

On a slave node, when an object is created with the **EXPORT** option, pSOS+m enters its name and ID in the Global Object Table on the object's node of residence. In addition, pSOS+m passes the object's name and ID to the master node for entry in the master node's Global Object Table.

Thus, every global object located on a slave node has entries in two Global Object Tables -- the one on its node of residence, and the one on the master node. On the master node, when an object is created with the **EXPORT** option, the global object's name and ID are simply entered in the master node's Global Object Table.

Similar operations occur when a global object is deleted. When a global object is deleted, it is removed from the master node's Global Object Table and its own node's Global Object Table if the object resides on a slave node.

The maximum number of objects (of all types) that can be exported is specified by the Multiprocessor Configuration Table entry, **mc_nglbobj**. During pSOS+m initialization, this entry is used to pre-allocate storage space for the Global Object Table. Note that the master node's Global Object Table is always much larger than Global Object Tables on slave nodes.

Formulae for calculating the sizes and memory usage of Global Object Tables are provided in the *pSOSystem Programmer's Reference.*

3.4.4 Ident Operations on Global Objects

The pSOS+m Object Ident system calls (e.g. **t_ident** or **q_ident**) perform run-time binding by converting an object's name into the object's ID. This may require searching the object tables on the local node and/or the Global Object Table on the master node. To search the master node's Global Object Table, slave nodes must post an **IDENT** request to the master node. On receiving this request, pSOS+m on the master node searches its Global Object Table and replies to the slave node with the object's ID, or an indication that the object does not exist.

Because objects created and exported by different nodes may not have unique names, the result of this binding may depend on the order and manner in which the object tables are searched. The table search order may be modified using the **node** input parameter to the Object Ident system calls. In particular,

1. If **node** equals 0, pSOS+m first searches the Local Object Table and then the Global Object Table on the caller's node. If the object is not found, a request is posted to the master node, which searches its Global Object Table, beginning with objects exported by node number 1, then node 2, and so on.

2. If **node** equals the local node's node number, then pSOS+m searches the Global Object Table on the local node only.
3. If **node** is not equal to the local node number, a request is posted to the master node, which searches its Global Object Table for objects created and exported by the specified node.

Typically, object binding is a one-time only, non-time-critical operation executed as part of setting up the application or when adding a new object.

3.5 Remote Service Calls

When pSOS+m receives a system call whose target object ID indicates that the object does not reside on the node from which the call is made, pSOS+m will process the system call as a *remote service call* (RSC).

In general, an RSC involves two nodes. The source node is the node from which the system call is made. The destination node is the node on which the object of the system call resides. To complete an RSC, the pSOS+m kernels on both the source and destination nodes must carry out a sequence of well-coordinated actions and exchange a number of internode packets.

There are two types of RSC, synchronous and asynchronous. Each is described in the following sections.

3.5.1 Synchronous Remote Service Calls

A synchronous RSC occurs whenever any of the following pSOS+m service calls are directed to an object that does not reside on the local node:

as_send()	**ev_send()**
q_broadcast()	**q_vbroadcast()**
q_receive()	**q_vreceive()**
q_send()	**q_vsend()**
q_urgent()	**q_vurgent()**
pt_getbuf()	**pt_retbuf()**
sm_p()	**sm_v()**
t_getreg()	**t_setreg()**
t_resume()	**t_suspend()**
t_setpri()	

Consider what happens when a task calls **q_send** to send a message to a queue on another node:

1. On the source node, pSOS+m receives the call, deciphers the QID, and determines that this requires an RSC;
2. pSOS+m calls the Kernel Interface (KI) to get a packet buffer, loads the buffer with the **q_send** information, and calls the KI to send the packet to the destination node;
3. If the KI delivers the packet successfully, pSOS+m blocks the calling task, and then switches to run another task;
4. Meanwhile, on the destination node, its KI senses an incoming packet (typically from an ISR), and calls the pSOS+m Announce-Packet entry;
5. When the KI's ISR exits, pSOS+m calls the KI to receive the packet, deciphers its contents, and generates an internal **q_send** call to deliver the message to the resident target queue;
6. If the **q_send** call is successful, then pSOS+m uses the packet buffer it received in Step 5 to build a reply packet, and calls KI to send the packet to the source node;
7. If the KI delivers the reply packet successfully, pSOS+m simply executes a normal dispatch to return to the user's application;
8. Back on the source node, its KI senses an incoming packet (typically from an ISR), and calls the pSOS+m Announce-Packet entry;
9. When the KI ISR exits, pSOS+m calls the KI to receive the packet, deciphers its contents, recognizes that it is a normal conclusion of an RSC, returns the packet buffer, unblocks the calling task, and executes a normal dispatch to return to the application.

This example shows a completely normal operation. If there is any error or abnormal condition at any level, the results may vary from a system shutdown to a simple error code being returned to the caller.

Certain pSOS+m system calls are not supported as RSCs. Most of these are excluded because they can never be RSCs -- for instance, calls that can only be self-directed at the calling task (for example, **t_mode**, **ev_receive**, and **tm_wkafter**). **tm_set** and **tm_get** are not supported because they affect resources, in this case time, that are otherwise strictly local resources.

Some calls are excluded because their implementation as RSCs would have meant compromises in other important respects. At present, object creation and deletion calls are not supported, for performance and robustness reasons. Notice that every system call that may be useful for communication, synchronization and state control is included.

Furthermore, note that RSCs are supported only if they are called from tasks. Calls from ISRs are illegal since the overhead associated with internode communication makes it unacceptable for use from an ISR.

In summary, in the event of an RSC, pSOS+m on the source and destination nodes use their respective KI to exchange packets which, in a manner completely transparent to the user's application, "bridge the gap" between the two nodes.

3.5.2 Asynchronous Remote Service Calls

When a task makes a synchronous remote service call, the task is blocked until a reply is received from the destination node. This allows errors and return values to be returned to the calling task and is essential to transparent operation across nodes. However, some service calls such as **q_send()** return only an error code and if the caller knows an error is not possible, then waiting for a reply needlessly delays execution of the calling task and consumes CPU resources with the processing of two context switches, as the task blocks and then unblocks.

For faster operation in these cases, pSOS+m offers asynchronous versions for the following pSOS+ system calls:

pSOS+ Synchronous Service	pSOS+m Asynchronous Call
q_send()	**q_asend()**
q_urgent()	**q_aurgent()**
q_vsend()	**q_avsend()**
q_vurgent()	**q_avurgent()**
sm_v()	**sm_av()**
ev_send()	**ev_asend()**

Asynchronous calls operate like their synchronous counterparts, except that the calling task does not wait for a reply and the destination node does not generate one.

An asynchronous RSC should be used only when an error is not expected. If an error occurs, however, pSOS+m on the destination node will send a packet to the source node describing the error. Since the state of the calling task is unknown (e.g. it may have been deleted), the source pSOS+m does not attempt to directly notify the calling task. Instead, it checks for a user-provided callout routine by examining the Multiprocessor Configuration Table entry **mc_asyncerr**. If provided, this routine is called.

What the **mc_asyncerr** callout routine does is up to the user. However, a normal sequence of events is to perform further error analysis and then shut down the node with a **k_fatal()** call. Other alternatives are to delete or restart the calling task, send an ASR or event to the calling task, or ignore the error altogether.

If an **mc_asyncerr** routine is not provided (**mc_asyncerr = 0**), pSOS+m generates an internal fatal error.

Note that an asynchronous service may operate on a local object. In this case, the call is performed synchronously since all relevant data structures are readily available. Nonetheless, should an error occur, it is handled as if the object were remote. Thus, **mc_asyncerr** is invoked and no error indication is returned to the caller. This provides consistent behavior regardless of the location of the referenced object.

Asynchronous calls are only supported in pSOS+m. If called when using pSOS+ (the single processor version), an error is returned.

3.5.3 Agents

Certain RSCs require waiting at an object on a remote node. For example, **q_receive** and **sm_p** may require the calling task to wait for a message or semaphore, respectively. If the message queue or semaphore is local, then pSOS+m simply enqueues the calling task's TCB to wait at the object. What if the object is not local?

Suppose the example in section 3.5.1 involves a **q_receive**, not a **q_send**, call. The transaction sequence is identical, up to when the destination node's pSOS+m deciphers the received packet, and recognizes the **q_receive**. pSOS+m uses a pseudo-object, called an *Agent*, to generate

the **q_receive** call to the target queue. If the queue is empty, then the Agent's Control Block, which resembles a mini-TCB, will be queued at the message wait queue. The destination node then executes a normal dispatch and returns to the application.

Later, when a message is posted to the target queue, the Agent is simply dequeued from the message wait queue. pSOS+m uses the original RSC packet buffer to hold a reply packet containing among other things the received message; it then calls the KI to send the reply packet back to the source node. The Agent is released to the free Agent pool, and all remaining transactions are again identical to that for **q_send**.

In summary, Agents are used to wait for messages or semaphores on behalf of the task that made the RSC. They are needed because the calling tasks are not resident on the destination node, and thus not available to perform any waiting function.

The Multiprocessor Configuration Table entry, **mc_nagent**, specifies the number of Agents that pSOS+m will allocate for that node. Since one Agent is used for every RSC that requires waiting on the destination node, this parameter must be large to support the expected worst case number of such RSCs.

3.5.4 RSC Overhead

In comparison to a system call whose target object is resident on the node from which the call is made, an RSC requires several hidden transactions between pSOS+m and the KI both on the source and destination nodes, not to mention the packet transit times. The exact measure of this overhead depends largely on the connection medium between the source and destination nodes.

If the medium is a memory bus, the KI operations will be quite fast, as is the packet transit time. On the other hand, if the medium is a network, especially one that uses a substantial protocol, the packet transit times may take milliseconds or more.

3.6 System Startup and Coherency

The master node must be the first node started in a pSOS+m multiprocessor system. After the master node is up and running, other nodes may then join. A slave node should not attempt to join until the master node is operational and it is the user's responsibility to ensure that this is the case. In a system in which several nodes are physically started at the same time (for example when power is applied to a VME card cage) this is easily accomplished by inserting a small delay in the startup code on the slave nodes. Alternately, the **ki_init** service can delay returning to pSOS+m until it detects that the master node is properly initialized and operational.

Slave nodes may join the system any time after the master node is operational. Joining requires no overt action by application code running on the slave node . pSOS+m automatically posts a join request to the master node during its initialization process. On the master node, pSOS+m first performs various coherency checks to see if the node should be allowed to join (see below) and if so, grants admission to the new node. Finally, it notifies other nodes in the system that the new node has joined.

For a multiprocessor pSOS+m system to operate correctly, the system must be coherent. That is, certain Multiprocessor Configuration Table parameters must have the same value on every node in the system. In addition, the pSOS+m versions on each node must be compatible. There are four important coherency checks that are performed whenever a slave node joins:

1. The pSOS+m version on each slave node must be compatible with the master node.
2. The maximum number of nodes in the system as specified in the Multiprocessor Configuration Table entry **mc_nnode** must match the value specified on the master node.
3. The maximum number of global objects on the node as specified by the Multiprocessor Configuration Table entry **mc_nglbobj** must match the value specified on the master node.
4. The maximum packet size that can be transmitted by the KI as specified by the Multiprocessor Configuration Table entry **mc_kimaxbuf** must match the value specified on the master node.

All of the above conditions are checked by the master node when a slave node attempts to join. If any condition is not met, the slave node will not be allowed to join. The slave node then aborts with a fatal error.

Joining nodes must observe one important timing limitation. In networks with widely varying transmission times between nodes, it is possible for a node to join the system, obtain the ID of an object on a remote node and post an RSC to that object, all before the object's node of residence has been notified that the new node has joined. When this occurs, the destination node simply ignores the RSC. This may cause the calling task to hang or, if the call was asynchronous, to proceed believing the call was successful.

To prevent such a condition, a newly joining node must not post an RSC to a remote node until a sufficient amount of time has elapsed to ensure the remote node has received notification of the new node's existence.

In systems with similar transmission times between all master and slave nodes, no special precautions are required, since all slaves would be informed of the new node well before the new node could successfully **IDENT** the remote object and post an RSC.

In systems with dissimilar transmission times, an adequate delay should be introduced in the **ROOT** task. The delay should be roughly equal to the worst case transmission time from the master to a slave node.

3.7 Node Failures

As mentioned before, the master node must never fail. In contrast, slave nodes may exit a system at any time. Although a node may exit for any reason, it is usually a result of a hardware or software failure. Therefore, this manual refers to a node that stops running for any reason as a *failed node*.

The failure of a node may have an immediate and substantial impact on the operation of remaining nodes. For example, nodes may have RSCs pending on the failed node, or there may be agents waiting on behalf of the failed node. As such, when a node fails, all other nodes in the system must be notified promptly, so corrective action can be taken.

The following paragraphs explain what happens when a node fails or leaves a system. In general, the master node is responsible for coordinating the graceful removal of a failed node. There are three ways that a master may learn of a node failure:

1. pSOS+m on the failing node internally detects a fatal error condition, which causes control to pass to its fatal error handler. The fatal error handler notifies the master and then shuts itself down (as described in Chapter 2, "pSOS+ Real-Time Kernel").

2. An application calls **k_fatal()** (without the **k_global** attribute). On a slave node, control is again passed to the pSOS+m internal fatal error handler, which notifies the master and then shuts itself down.

3. An application on any node (not necessarily the failing node) calls **k_terminate()**, which notifies the master.

Upon notification of a node failure, the master does the following:

1. First, if notification did not come from the failed node, the master sends a shutdown packet to the failed node. If the failed node receives it (that is, it has not completely failed yet), it performs the shutdown procedure as described in Chapter 2, "pSOS+ Real-Time Kernel."

2. Second, it sends a failure notification packet to all remaining slave nodes.

3. Lastly, it removes all global objects created by the failed node from its global object table.

pSOS+m on all nodes, including the master, perform the next 4 steps after receiving notification of a node failure:

1. pSOS+m calls the Kernel Interface (KI) service **ki_roster** to notify the KI that a node has left the system.

2. pSOS+m calls the user-provided routine pointed to by the Multiprocessor Configuration Table entry **mc_roster** to notify the application that a node has left the system.

3. All agents waiting on behalf of the failed node are recovered.

4. All tasks waiting for RSC reply packets from the failed node are awakened and given error **ERR_NDKLD**, indicating that the node failed while the call was in progress.

After all of the above steps are completed, unless notified by your **mc_roster** routine, it is possible that your application code may still use object IDs for objects that were on the failed node. If this happens, pSOS+m returns the error **ERR_STALEID**.

3.8 Slave Node Restart

A node that has failed may subsequently restart and rejoin the system. pSOS+m treats a rejoining node exactly like a newly joining node, that is, as described in section 3.6. In fact, internally, pSOS+m does not distinguish between the two cases. However, a rejoining node introduces some special considerations that are discussed in the following subsections.

3.8.1 Stale Objects and Node Sequence Numbers

Recall from section 3.7 that when a node exits, the system IDs for objects on the node may still be held by task level code. Such IDs are called *stale IDs*. So long as the failed node does not rejoin, detection of stale IDs is trivial since the node is known not to be in the system. However, should the failed node rejoin, then, in the absence of other protection mechanisms, a stale ID could again become valid. This might lead to improper program execution.

To guard against use of stale IDs after a failed node has rejoined, every node is assigned a *sequence number*. The master node is responsible for assigning and maintaining sequence numbers. A newly joining node is assigned sequence number = 1 and the sequence number is incremented thereafter each time the node rejoins. All object IDs contain both the node number and sequence number of the object's node of residence. Therefore, a stale ID is easily detected by comparing the sequence number in the ID to the current sequence number for the node.

Object IDs are 32-bit unsigned integers. Because only 32 bits are available in a node number, the number of bits used to encode the sequence number depends on the maximum number of nodes in the system as specified in the Multiprocessor Configuration Table entry **mc_nnode**. If **mc_nnode** is less than 256, then 8 bits are used to encode the sequence number and the maximum sequence number is 255. If **mc_nnode** is greater than or equal to 256, then the number of bits used to encode the sequence number is given by the formula

$$\mathbf{16 - ceil(\log_2(mc\_nnode + 1))}$$

For example, in a system with 800 nodes, 6 bits would be available for the sequence number and the maximum sequence number would therefore be 63. In the largest possible system (recall **mc_nnode** may not exceed 16383), there would be 2 bits available to encode the sequence number.

Once a node's sequence number reaches the maximum allowable value, the next time the node attempts to rejoin, the action taken by pSOS+m depends on the value of the Multiprocessor Configuration Table entry **mc_flags** *on the rejoining slave node.* If the **SEQWRAP** bit is not set, then the node will not be allowed to rejoin. However, if **SEQWRAP** is set, then the sequence number will wrap around to one. Since this could theoretically allow a stale ID to be reused, this option should be used with caution.

3.8.2 Rejoin Latency Requirements

When a node fails, considerable activity occurs on every node in the system to ensure that the node is gracefully removed from the system. If the node should rejoin too soon after failing, certain inter-nodal activities by the new instantiation of the node may be mistakenly rejected as relics of the old instantiation of the node.

To avoid such errors, a failed node must not rejoin until all remaining nodes have been notified of the failure and have completed the steps described in section 3.7. In addition, there must be no packets remaining in transit in the KI, either to or from the failed node, or reporting failure of the node, or awaiting processing at any node. This is usually accomplished by inserting a small delay in the node's initialization code. For most systems communicating through shared memory, a delay of 1 second should be more than adequate.

3.9 Global Shutdown

A global shutdown is a process whereby all nodes stop operating at the same time. It can be caused for two reasons:

1. A fatal error occurred on the master node.
2. A **k_fatal()** call was made with the **global** attribute set. In this case, the node where the call was made notifies the master node.

In either case, the master node then sends every slave node a shutdown packet. All nodes then perform the normal pSOS+m shutdown procedure.

3.10 The Node Roster

On every node, pSOS+m internally maintains an up-to-date node roster at all times, which indicates which nodes are presently in the system. The roster is a bit map encoded in 32-bit (long word) entries. Thus, the first long word contains bits corresponding to nodes 1 - 32, the second nodes 33 - 64, etc. Within a long word, the rightmost (least significant) bit corresponds to the lowest numbered node.

The map is composed of the minimum number of long words needed to encode a system with **mc_nnode**, as specified in the Multiprocessor Configuration Table. Therefore, some bits in the last long word may be unused.

Application code and/or the KI may also need to know which nodes are in the system. Therefore, pSOS+m makes its node roster available to both at system startup and keeps each informed of any subsequent roster changes. The application is provided roster information via the user-provided routine pointed to by the Multiprocessor Configuration Table entry **mc_roster**. The KI is provided roster information via the KI service **ki_roster**. For more information on KI service calls or the Multiprocessor Configuration Table, see the *pSOSystem Programmer's Reference.*

3.11 Dual-Ported Memory Considerations

Dual-ported memory is commonly used in memory-bus based multiprocessor systems. However, it poses several unique problems to the software: any data structure in dual-ported memory has two addresses, one for each port. Consider the problem when one processor node passes the address of a data structure to a second node. If the data structure is in dual-ported memory, the address may have to be translated before it can be used by the target node, depending on whether or not the target node accesses this memory through the same port as the sender node.

To overcome this confusion over the duality of address and minimize its impact on user application code, pSOS+m includes facilities that perform address conversions. But first, a few terminology definitions.

3.11.1 P-Port and S-Port

A zone is a piece of contiguously addressable memory, which can be single or dual ported. The two ports of a dual-ported zone are named the *p-port* and the *s-port*. The (private) p-port is distinguishable in that it is typically reserved for one processor node only. The (system) s-port is normally open to one or more processor nodes.

In a typical pSOS+m configuration, the multiple nodes are tied via a system bus, e.g. VME or Multibus. In this case, each dual-ported zone's s-port would be interfaced to the system bus, and each p-port would be connected to one processor node via a private bus that is usually, but not necessarily, on the same circuit board.

If a node is connected to the p-port of a dual-ported zone, then three entries in its pSOS+m Multiprocessor Configuration Table must be used to describe the zone. **mc_dprext** and **mc_dprint** specify the starting address of the zone, as seen from the s-port and the p-port, respectively. **mc_dprlen** specifies the size of the zone, in bytes. In effect, these entries define a special window on the node's address space. pSOS+m uses these windows to perform transparent address conversions for the user's application.

If a node is not connected to any dual-ported zone, or accesses dual-ported zones only through their s-ports, then the three configuration table entries should be set to 0. Notice that the number of zones a processor node can be connected to via the p-port is limited to one.

NOTE: A structure (user or pSOS+m) must begin and end in a dual port zone. It must not straddle a boundary between single and dual ported memory.

3.11.2 Internal and External Address

When a node is connected to a dual-ported zone, any structure it references in that zone, whether it is created by the user's application code or by pSOS+ (e.g. a partition buffer), is defined to have two addresses:

1. The internal address is defined as the address used by the node to access the structure. Depending on the node, this may be the p-port or the s-port address for the zone.
2. The external address is always the s-port address.

3.11.3 Usage Within pSOS+m Services

Any address in a dual ported zone used as input to pSOS+m or entered in a Configuration Table must be an internal address (to the local node). Similarly, when a pSOS+m system call outputs an address that is in a dual ported zone, it will always be an internal address to the node from which the call is made.

Consider in particular a partition created in a dual ported zone and exported to enable shared usage by two or more nodes. A **pt_getbuf** call to this partition automatically returns the internal address of the allocated buffer. In other words, pSOS+m always returns the address that the calling program can use to access the buffer. If the calling node is tied to the zone's p-port, then the returned internal address will be the p-port address. If the calling node is tied to the s-port, then the returned internal address will be the s-port address.

3.11.4 Usage Outside pSOS+

Often, operations in dual-ported zones fall outside the context of pSOS+. For example, the address of a partition buffer or a user structure may be passed from one node to another within the user's application code. If this address is in a dual ported zone, then the two system calls, **m_int2ext** and **m_ext2int**, may need to be used to perform a necessary address conversion.

Observe the following simple rule:

When an address within a dual-port zone must be passed from one node to another, then pass the external address.

The procedure is quite simple. Since the sending node always knows the internal address, it can call **m_int2ext** to first convert it to the external address. On the receiving node, **m_ext2int** can be used to convert and obtain the internal address for that node.

4 Network Programming

4.1 Overview of Networking Facilities

pSOSystem provides an extensive set of networking facilities for addressing a wide range of interoperability and distributed computing requirements. These facilities include

TCP/IP Support - pSOSystem's TCP/IP networking capabilities are constructed around the pNA+ software component. pNA+ includes TCP, UDP, IP, ICMP, and ARP accessed through the industry standard socket programming interface. pNA+ offers services to application developers as well as to other pSOSystem networking options such as RPC, NFS, FTP, and so forth.

In addition, pNA+ supports the Management Information Base for Network Management of TCP/IP-based Internets (MIB-II) standard. pNA+ also works in conjunction with pSOSystem's cross development tools to provide a network-based download and debug environment for single- or multi- processor target systems.

SNMP - Simple Network Management Protocol, is a standard used for managing TCP/IP networks and network devices. Because of its flexibililty and availability, SNMP has become the most viable way to manage large, heterogeneous networks containing commercial or custom devices.

FTP, Telnet, TFTP - pSOSystem includes support for the well known internet protocols FTP and Telnet (client and server side), and TFTP. FTP client allows you to transfer files to and from

remote systems. FTP server allows remote users to read and write files from and to pHILE+ managed devices. Telnet client enables you to login to remote systems, while Telnet server offers login capabilities to pSOSystem's shell, pSH, from remote systems. TFTP is used in pSOSystem Boot ROMs and is normally used to boot an application from a network device.

RPCs - pSOSystem fully supports Sun Microsystem's Remote Procedure Call (RPC) and eXternal Data Representation (XDR) specifications via the pRPC+ software component. pRPC+ allows you to construct distributed applications using the familiar C procedure call paradigm. With pRPC+, pSOS+ tasks and UNIX processes can invoke procedures for execution on other pSOSystem or UNIX machines.

NFS - pSOSystem offers both NFS client and NFS server support. NFS server allows remote systems to access files stored on pHILE+ managed devices. NFS client facilities are part of pHILE+ and allow your application to transparently access files stored on remote storage devices.

STREAMS - is an extremely flexible facility for developing system communication services. It can be used to implement services ranging from complete networking protocol suites to individual device drivers. Many modern networking protocols, including Windows NT and UNIX System V Release 4.2 networking services, are implemented in a STREAMS environment. pSOSystem offers a complete System V Release 4.2 -compatible STREAMS environment called OpEN (Open Protocol Embedded Networking).

The following documents published by Prentice Hall provide more detailed information on UNIX System V Release 4.2:

- *Operating System API Reference* (ISBN# 0-13-017658-3)
- *STREAMS Modules and Drivers* (ISBN# 0-13-066879-6)
- *Network Programming Interfaces* (ISBN# 0-13-017641-9)
- *Device Driver Reference* (ISBN# 0-13-042631-8)

X Windows Support - pSOSystem contains a complete implementation of MIT's X Windows client-side Xlib library via the pX11+ component. pX11+ allows an application to display output and read input from any X server located on an Internet network.

This chapter describes the pNA+, pRPC+, and pX11+ network components. The FTP, Telnet, pSH, TFTP, and NFS server facilities are documented in the *pSOSystem Programmer's Reference* manual. NFS client services are described along with pHILE+ in Chapter 5, "pHILE+ File System Manager."

Detailed information on SNMP is available in the *SNMP User's Manual*, and STREAMS is documented in the *OPEN User's Manual*, which describes pSOSystem's OPEN (Open Protocol Embedded Networking) product.

4.2 pNA+ Software Architecture

pNA+ is organized into four layers. Figure 7 illustrates the architecture and how the protocols fit into it.

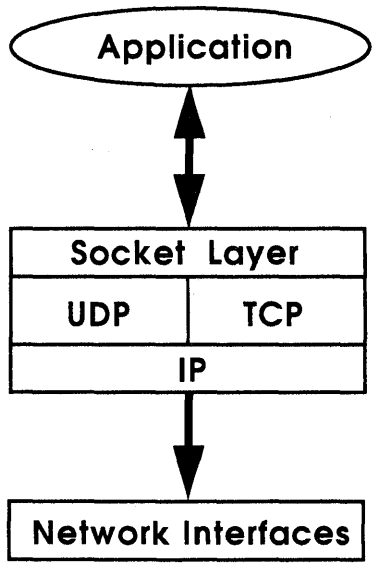

Figure 7. pNA+ Architecture

The *socket layer* provides the application programming interface. This layer provides services, callable as re-entrant procedures, which your application uses to access Internet protocols; it conforms to industry

standard UNIX 4.3 BSD socket syntax and semantics. In addition, this layer contains enhancements specifically for embedded real-time applications.

The *transport layer* supports the two Internet Transport protocols, Transmission Control Protocol (TCP) and User Datagram Protocol (UDP). These protocols provide network independent transport services. They are built on top of the Internet Protocol (IP).

TCP provides reliable, full-duplex, task-to-task data stream connections. It is based on the Internet layer, but adds reliability, flow control, multiplexing, and connections to the capabilities provided by the lower layers.

UDP provides a datagram mode of packet-switched communication. It allows users to send messages with a minimum of protocol overhead. However, ordered, reliable delivery of data is not guaranteed.

The IP layer is used for transmitting blocks of data called datagrams. This layer provides packet routing, fragmentation and reassembly of long datagrams through a network or internet.

The Network Interface (NI) layer isolates the IP layer from the physical characteristics of the underlying network medium. It is hardware dependent and is responsible for transporting packets within a single network. Because it is hardware dependent, the network interface is not part of pNA+ proper. Rather, it is a provided by the user, or by ISI as a separate piece of software.

In addition to the protocols described, pNA+ supports the Address Resolution Protocol (ARP) and the Internet Control Message Protocol (ICMP).

ICMP is used for error reporting and for other network-management tasks. It is layered above IP for input and output operations, but it is logically a part of IP, and is usually not accessed by users. See Section 4.14, "Internet Control Message Protocol (ICMP)."

ARP is used to map Internet addresses to physical network addresses; it is described in Section 4.10.2, "Address Resolution Protocol (ARP)."

4.3 The Internet Model

pNA+ operates in an internet environment. An *internet* is an interconnected set of networks. Each constituent network supports communication among a number of attached devices or *nodes*. In addition, networks are connected by nodes that are called *gateways*. Gateways provide a communication path so that data can be exchanged between nodes on different networks.

Nodes communicate by exchanging *packets*. Every packet in transit through an internet has a *destination Internet address*, which identifies the packet's final destination. The source and destination nodes can be on the same network (i.e. *connected*), or they can be on different networks (i.e. *indirectly connected*). If they are on different networks, the packet must pass through one or more gateways.

4.3.1 Internet Addresses

Each node in an internet has at least one unique *Internet (IP) address*. An Internet address is a 32-bit number that begins with a network number, followed by a node number. There are three formats or classes of Internet addresses. The different classes are distinguished by their high-order bits. The three classes are defined as A, B and C, with high-order bits of 0, 10, and 110. They use 8, 16, and 24 bits, respectively, for the network part of the address. Each class has fewer bits for the node part of the adddress and thus supports fewer nodes than the higner classes.

Externally, an Internet address is represented as a string of four 8-bit values separated by dots. Internally, an Internet address is represented simply as a 32-bit value. For example, the internet address 90.0.0.1 is internally represented as 0x5a000001. This address identifies node 1 on network 90. Network 90 is a class A network.

In the networking literature, nodes are sometimes called *hosts*. However, in real-time systems, the term host is normally used to refer to a development system or workstation (as opposed to a target system). Therefore, we choose to use the term *node* rather than host.

Note that a node can have more than one Internet address. A gateway node, for example, is attached to at least two physical networks and therefore has at least two Internet addresses. Each Internet address corresponds to one node-network connection.

4.3.2 Subnets

As mentioned above, an Internet address consists of a network part and a host part. To provide additional flexibility in the area of network addressing, the notion of *subnet addressing* has become popular, and is supported by pNA+.

Conceptually, subnet addressing allows you to divide a single network into multiple sub-networks. Instead of dividing a 32-bit Internet address into a network part and host part, subnetting divides an address into a network part and a *local* part. The local part is then sub-divided into a sub-net part and a host part. The sub-division of the host part of the Internet address is transparent to nodes on other networks.

Sub-net addressing is implemented by extending the network portion of an Internet address to include some of the bits that are normally considered part of the host part. The specification as to which bits are to be interpreted as the network address is called the *network mask*. The network mask is a 32-bit value with ones in all bit positions that are to be interpreted as the network portion.

For example, consider a pNA+ node with an address equal to 128.9.01.01. This address defines a Class B network with a network address equal to 128.9. If the network is assigned a network mask equal to 0xffffff00, then, from pNA+'s perspective, the node resides on network 128.9.01.

A network mask can be defined for each Network Interface (NI) installed in your system.

4.3.3 Broadcast Addresses

pNA+ provides an optional broadcast capability, if the underlying network supports it. An Internet address with a node number (*i.e.* host part) consisting of either all ones or all zeros is designated a *broadcast address*. A broadcast address is used to refer to all of the nodes on a given network.

For example, you can broadcast a packet to all of the nodes on the Class B network 128.1 by sending a packet to address 128.1.255.255. Similarly the broadcast address of a node with IP address 128.1.200.1 and netmask of 0xffffff00 is 128.1.200.255.

4.3.4 A Sample Internet

Figure 8 depicts an internet consisting of two networks.

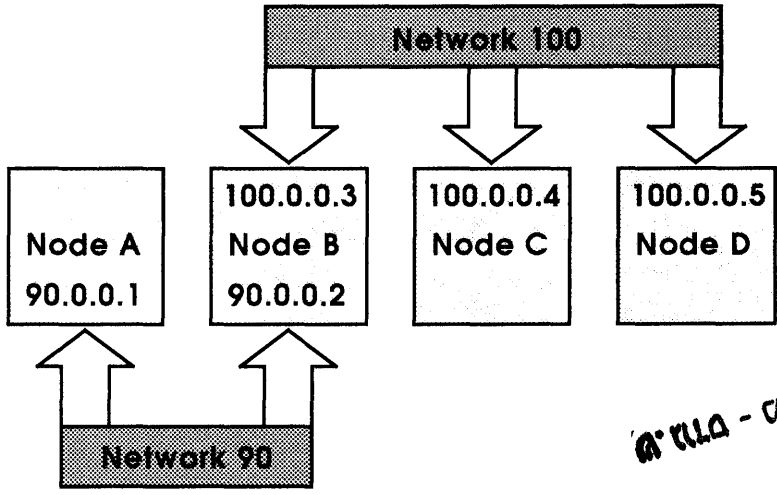

Figure 8. A Sample Internet

Note that since node B is on both networks, it has two Internet addresses and serves as a gateway between networks 90 and 100. For example, if node A wants to send a packet to node D, it sends the packet to node B, which in turn sends it to node D.

4.4 The Socket Layer

The socket layer is the programmer's interface to pNA+. It is based on the notion of sockets and designed to be syntactically and semantically compatible with UNIX 4.3 BSD networking services. This section is intended to provide a brief overview of sockets and how they are used.

4.4.1 Basics

A *socket* is an endpoint of communication. It is the basic building block for communication. Tasks communicate by sending and receiving data through sockets.

Sockets are typed according to the characteristics of the communication they support. pNA+ provides three types of sockets supporting three different types of service:

- *Stream* sockets use the Transmission Control Protocol (TCP) and provide a connection-based communication service. Before data is transmitted between stream sockets, a connection is established between them.
- *Datagram* sockets use the User Datagram Protocol (UDP) and provide a connectionless communication service. Datagram sockets allow tasks to exchange data with a minimum of protocol overhead. However, reliable delivery of data is not guaranteed.
- *Raw* sockets provide user level access to the IP and ICMP (see section 4.14) layers. This enables you to implement transport protocols (other than TCP/UDP) over the IP layer. They provide connectionless and datagram communication service.

4.4.2 Socket Creation

Sockets are created via the **socket()** system call. The type of the socket (stream, datagram, or raw) is given as an input parameter to the call. A *socket descriptor* is returned, which is then used by the creator to access the socket. An example of **socket()** used to create a stream socket is as follows:

```
s = socket (AF_INET, SOCK_STREAM, 0);
```

The returned socket descriptor can only be used by the socket's creator. However, the **shr_socket()** system call can be used to allow other tasks to reference the socket:

```
ns = shr_socket (s, tid);
```

The parameter **s** is a socket descriptor used by the calling task to reference an existing socket [**s** is normally a socket descriptor returned by **socket()**]. The parameter **tid** is the task ID of another task that wants to access the same socket. **shr_socket()** returns a new socket descriptor **ns**, which can be used by **tid** to reference the socket. This system call is useful when designing UNIX-style server programs.

4.4.3 Socket Addresses

Sockets are created without *addresses*. Until an address is assigned or *bound* to a socket, it cannot be used to receive data. A socket address consists of a user-defined 16-bit port number and a 32-bit Internet address. The socket address functions as a name that is used by other entities, such as tasks residing on other nodes within the internet, to reference the socket.

The **bind()** system call is used to bind a socket address to a socket. **bind()** takes as input a socket descriptor and a socket address and creates an association between the socket and the address specified. An example using **bind()** is as follows

```
bind (s, addr, addrlen);
```

4.4.4 Connection Establishment

When two tasks wish to communicate, the first step is for each task to create a socket. The next step depends on the type of sockets that were created. Most often stream sockets are used; in which case, a connection must be established between them.

Connection establishment is usually asymmetric, with one task acting as a *client* and the other task a *server*. The server binds an address (i.e. a 32-bit Internet address and a 16-bit port number) to its socket (as described above) and then uses the **listen()** system call to set up the socket, so that it can accept connection requests from clients. The **listen()** call takes as input a socket descriptor and a **backlog** parameter. **backlog** specifies a limit to the number of connection requests that can be queued for acceptance at the socket.

A client task can now initiate a connection to the server task by issuing the **connect()** system call. **connect()** takes a socket address and a socket descriptor as input. The socket address is the address of the socket at which the server is listening. The socket descriptor identifies a socket that constitutes the client's endpoint for the client-server connection. If the client's socket is unbound at the time of the **connect()** call, an address is automatically selected and bound to it.

In order to complete the connection, the server must issue the **accept()** system call, specifying the descriptor of the socket that was specified in the prior **listen()** call. The **accept()** call does not connect the initial socket, however. Instead, it creates a new socket with the same

properties as the initial one. This new socket is connected to the client's socket, and its descriptor is returned to the server. The initial socket is thereby left free for other clients that might want to use **connect()** to request a connection with the server.

If a connection request is pending at the socket when the **accept()** call is issued, a connection is established. If the socket does not have any pending connections, the server task blocks, unless the socket has been marked as non-blocking (see section 4.4.9), until such time as a client initiates a connection by issuing a **connect()** call directed at the socket.

Although not usually necessary, either the client or the server can optionally use the **getpeername()** call to obtain the address of the *peer* socket, that is, the socket on the other end of the connection.

The following illustrates the steps described above.

SERVER	CLIENT
`socket(domain, type, protocol);`	`socket(domain, type, protocol);`
`bind(s, addr, addrlen);`	`listen(s, backlog);`
`accept(s, addr, addrlen);`	`connect(s, addr, addrlen);`

4.4.5 Data Transfer

After a connection is established, data can be transferred. The **send()** and **recv()** system calls are designed specifically for use with sockets that have already been connected. The syntax is as follows:

```
send(s, buf, buflen, flags);

recv(s, buf, buflen, flags);
```

A task sends data through the connection by calling the **send()** system call. **send()** accepts as input a socket descriptor, the address and length of a buffer containing the data to transmit, and a set of flags. A flag can be set to mark the data as "out-of-band," that is, high-priority, so that it can receive special handling at the far end of the connection. Another flag can be set to disable the routing function for the data; that is, the data will be dropped if it is not destined for a node that is directly connected to the sending node.

The socket specified by the parameter **s** is known as the *local* socket, while the socket at the other end of the connection is called the *foreign* socket.

When **send()** is called, pNA+ copies the data from the buffer specified by the caller into a send buffer associated with the socket and attempts to transmit the data to the foreign socket. If there are no send buffers available at the local socket to hold the data, **send()** blocks, unless the socket has been marked as non-blocking. The size of a socket's send buffers can be adjusted with the **setsockopt()** system call.

A task uses the **recv()** call to receive data. **recv()** accepts as input a socket descriptor specifying the communication endpoint, the address and length of a buffer to receive the data, and a set of flags. A flag can be set to indicate that the **recv()** is for data that has been marked by the sender as out-of-band only. A second flag allows **recv()** to "peek" at the message; that is, the data is returned to the caller, but not consumed.

If the requested data is not available at the socket, and the socket has not been marked as non-blocking, **recv()** causes the caller to block until the data is received. On return from the **recv()** call, the server task will find the data copied into the specified buffer.

4.4.6 Connectionless Sockets

While connection-based communication is the most widely used paradigm, connectionless communication is also supported via datagram or raw sockets. When using datagram sockets, there is no requirement for connection establishment. Instead, the destination address (i.e the address of the foreign socket) is given at the time of each data transfer.

To send data, the **sendto()** system call is used:

```
sendto(s, buf, buflen, flags, to, tolen);
```

The **s**, **buf**, **buflen**, and **flags** parameters are the same as those in **send()**. The **to** and **tolen** values are used to indicate the address of the foreign socket that will receive the data.

The **recvfrom()** system call is used to receive data:

```
recvfrom(s, buf, buflen, flags, to, tolen);
```

The address of the data's sender is returned to the caller via the **to** parameter.

4.4.7 Discarding Sockets

Once a socket is no longer needed, its socket descriptor can be discarded by using the **close()** system call. If this is the last socket descriptor associated with the socket, then **close()** de-allocates the socket control block (see section 4.4.11) and, unless the LINGER option is set (see section 4.4.8), discards any queued data. As a special case, **close(0)** closes all socket descriptors that have been allocated to the calling task. This is particularly useful when a task is to be deleted.

4.4.8 Socket Options

The **setsockopt()** system call allows a socket's creator to associate a number of options with the socket. These options modify the behavior of the socket in a number of ways, such as whether messages sent to this socket should be routed to networks that are not directly connected to this node (the **DONTROUTE** option); whether sockets should be deleted immediately if their queues still contain data (the **LINGER** option); whether packet broadcasting is permitted via this socket (the **BROADCAST** option), and so forth. A detailed description of these options and their effects is given in the *pSOSystem Programmer's Reference* manual.

Options associated with a socket can be checked via the **getsockopt()** system call.

4.4.9 Non-Blocking Sockets

Many socket operations cannot be completed immediately. For instance, a task might attempt to read data that is not yet available at a socket. In the normal case, this would cause the calling task to block until the data became available. A socket can be marked as non-blocking through use of the **ioctl()** system call. If a socket has been marked as non-blocking, an operation request that cannot be completed without blocking does not execute and an error is returned to the caller.

The **select()** system call can be used to check the status of a socket, so that a system call will not be made that would cause the caller to block.

4.4.10 Out-of-Band Data

Stream sockets support the notion of out-of-band data. Out-of-band data is a logically independent transmission channel associated with each pair of connected sockets. The user has the choice of receiving out-of-band data either in sequence with the normal data or independently of the normal sequence. It is also possible to "peek" at out-of-band data. A logical mark is placed in the data stream to indicate the point at which out-of-band data was sent.

If multiple sockets might have out-of-band data awaiting delivery, for exceptional conditions **select()** can be used to determine those sockets with such data pending.

To send out-of-band data, the **MSG_OOB** flag should be set with the **send()** and **sendto()** system calls. To receive out-of-band data, the **MSG_OOB** flag is used when calling **recv()** and **recvfrom()**. The **SIOCATMARK** option in the **ioctl()** system call can be used to determine if out-of-band data is currently ready to be read.

4.4.11 Socket Data Structures

pNA+ uses two data structures to manage sockets: *socket control blocks* and *open socket tables.*

A socket control block (SCB) is a system data structure used by pNA+ to maintain state information about a socket. During initialization, pNA+ creates a fixed number of SCBs. An SCB is allocated for a socket when it is created via the **socket()** call.

Every task has an open socket table associated with it. This table is used to store the addresses of the socket control blocks for the sockets that can be referenced by the task. A socket descriptor is actually an index into an open socket table. Since each task has its own open socket table, you can see that one socket might be referenced by more than one socket descriptor. New socket descriptors for a given socket can be obtained with the **shr_socket()** system call (see section 4.4.2).

4.5 The pNA+ Daemon Task

When pNA+ system calls are made, there are three possible outcomes:

1. pNA+ executes the requested service and returns to the caller.
2. The system call cannot be completed immediately, but it does not require the caller to wait. In this case, pNA+ schedules the necessary operations and returns control to the caller. For example, the **send()** system call copies data from the user's buffer to an internal buffer. The data might not actually be transmitted until later, but control returns to the calling task, which continues to run.
3. The system call cannot be completed immediately and the caller must wait. For example the user might attempt to read data that is not yet available. In this case, pNA+ blocks the calling task. The blocked task is eventually rescheduled by subsequent asynchronous activity.

As the above indicates, the Internet protocols are not always synchronous. That is, not all pNA+ activities are initiated directly by a call from an application task. Rather, certain "generic" processing activities are triggered in response to external events such as incoming packets and timer expirations. To handle asynchronous operations, pNA+ creates a daemon task called **pNAD**.

pNAD is created during pNA+ initialization. It is created with a priority of 255 to assure its prompt execution. The priority of **pNAD** can be lowered with the pSOS+ **t_setpri** call. However, its priority must be higher than the priority of any task calling pNA+.

pNAD is normally blocked, waiting for one of two events, encoded in bits 30 and 31. When **pNAD** receives either of these two events, it is unblocked and preempts the running task.

The first event (bit 31) is sent to **pNAD** by pNA+ upon receipt of a packet when the pNA+ **ANNOUNCE_PACKET** entry is called, either by an ISR or **ni_poll**. Based on the content of the packet, **pNAD** takes different actions, such as waking up a blocked task, sending a reply packet, or, if this is a gateway node, forwarding a packet. The last action should be particularly noted; that is, if a node is a gateway, **pNAD** is responsible for forwarding packets. If the execution of **pNAD** is inhibited or delayed, packet routing will also be inhibited or delayed.

The second event (bit 30) is sent every 100 milliseconds as a result of a pSOS+ **tm_evevery** system call. When **pNAD** wakes up every 100ms, it performs time-specific processing for TCP that relies heavily on time-related retries and timeouts. After performing its time-related processing, **pNAD** calls **ni_poll** for each Network Interface that has its **POLL** flag set.

4.6 The User Signal Handler

pNA+ defines a set of signals, which correspond to unusual conditions that might arise during normal execution. The user can provide an optional signal handler, which is called by pNA+ when one of these "unusual" or unpredictable conditions occur. For example, if urgent data is received, or if a connection is broken, pNA+ calls the user-provided signal handler.

The address of the user-provided signal handler is provided in the pNA+ Configuration Table entry **NC_SIGNAL**. When called by pNA+, the handler receives as input the signal type (i.e. the reason the handler is being called), the socket descriptor of the affected socket, and the TID of the task that "owns" the affected socket. When a socket is first created, it has no owner; it must be assigned one using the **ioctl()** system call.

It is up to the user to decide how to handle the signal. For example, the handler can call the pSOS+ **as_send** system call to modify the execution path of the owner. A user signal handler is not required. The user can choose to ignore signals generated by pNA+ by setting **NC_SIGNAL** equal to zero. In addition, if the socket has no "owner," the signals are dropped. The signals are provided to the user so that the application can respond to these unpredictable conditions, if it chooses to do so.

The following is a list of the signals that can be generated by pNA+:

SIGIO	0x40000000	I/O activity on the socket
SIGPIPE	0x20000000	Connection has been disconnected
SIGURG	0x10000000	Urgent data has been received

The *pSOSystem Programmer's Reference* manual describes the calling conventions used by pNA+ when calling the user-provided signal handler.

4.7 Error Handling

pNA+ uses the UNIX BSD 4.3 socket level error reporting mechanisms. When UNIX detects an error condition, it stores an error code into the internal variable **errno** and returns -1 to the caller. To get the error code, the calling task simply reads **errno** prior to making another system call.

pNA+ implements the UNIX error reporting mechanisms, but with two variations. First, pNA+ uses a function rather than a variable to return the error code. Second, the corresponding pNA+ function is named **pna_errno** instead of **errno**. Thus, if pNA+ returns an error code of -1, your application should call **pna_errno()** to determine the exact cause of the error.

4.8 Packet Routing

pNA+ includes complete routing facilities. This means that, in addition to providing end-to-end communication between two network nodes, a pNA+ node forwards packets in an Internet environment. When pNA+ receives a packet addressed to some other node, it attempts to forward the packet toward its destination.

pNA+ forwards packets based on *routes* that define the connectivity between nodes. A route provides reachability information by defining a mapping between a destination address and a next hop within a physically attached network.

Routes can be classified as either *direct* or *indirect*. A *direct route* defines a path to a directly connected node. Packets destined for that node are sent directly to the final destination node. An *indirect route* defines a path to an indirectly connected node (see section 4.3). Packets addressed to an indirectly connected node are routed through an intermediate gateway node.

Routes can be classified further as either *host* or *network*. A *host route* specifies a path to a particular destination node, based on the complete destination node's IP address. A *network route* specifies a path to a destination node, based only on the network portion of the destination node's IP address. That is, a network route specifies a path to an entire destination network, rather than to a particular node in the network.

Direct routes provide a mapping between a destination address and a Network Interface (NI). They are added during NI initialization. When an NI is added into the system (see section 4.9.6), pNA+ adds a direct route for that NI. If the network is a point-to-point network, a pNA+ node is connected to a single node (see section 4.9.5), and the route is a host route. Otherwise, it is a network route.

Indirect routes provide a mapping between a destination address and a gateway address. Unlike direct routes, indirect routes are not created automatically by pNA+. Indirect routes are created explicitly, either by entries in the pNA+ Configuration Table, or by using the pNA+ system calls **chng_route()** or **ioctl()**.

pNA+ supports one final routing mechanism, a *default gateway*, which can be specified in the pNA+ configuration table. The default gateway specifies the address to which all packets are forwarded when no other route for the packet can be found. In fact, in most pNA+ installations, a default route is the only routing information ever needed.

In summary, pNA+ uses the following algorithm to determine a packet route:

1. pNA+ first looks for a host route using the destination node's complete IP address. If one exists and is a direct route, the packet is sent directly to the destination node. If it is an indirect route, the packet is forwarded to the gateway specified in the route.
2. If a host route does not exist, pNA+ looks for a network route using the network portion of the destination node's IP address. If one exists and is a direct route, the packet is sent directly to the destination node. If it is an indirect route, the packet is forwarded to the gateway specified in the route.
3. If a network route does not exist, pNA+ forwards the packet to the default gateway, if one has been provided.
4. Otherwise, the packet is dropped.

Routes can be configured into pNA+ during initialization. The configuration table entry **NC_IROUTE** contains a pointer to an Initial Routing Table (see the *pSOSystem Programmer's Reference* manual). They can also be added or altered dynamically, using the pNA+ function calls **chng_route()** and **ioctl()**. For simplicity, most systems use a default gateway node. A default gateway is specified by the configuration table entry **NC_DEFGN**.

4.9 Network Interfaces

pNA+ accesses a network by calling a user-provided layer of software called the Network Interface (NI). The interface between pNA+ and the NI is standard and independent of the network's physical media or topology; it isolates pNA+ from the network's physical characteristics.

The NI is essentially a device driver that provides access to a transmission medium. (The terms *network interface, NI,* and *network driver* are all used interchangeably in this manual.) A detailed description of the interface between pNA+ and the NI is given in the *pSOSystem Programmer's Reference* manual.

There must be one NI for each network connected to a pNA+ node. In the simplest case, a node is connected to just one network and will have just one NI. However, a node can be connected to several networks simultaneously and therefore have several network interfaces. Each NI is assigned a unique IP address.

Each network connection (NI) has a number of attributes associated with it. They are as follows:

- The address of the NI entry point
- The IP address
- The maximum transmission unit
- The length of its hardware address
- Control flags
- The network mask
- Destination IP address (point-to-point links)

pNA+ stores these attributes for all of the network interfaces installed in your system in the NI Table, discussed in Section 4.9.6, "The NI Table." NI attributes can be modified using **ioctl()**. The first two attributes are self-explanatory. Maximum transmission units, hardware addresses, control flags, network subnet mask, and destination IP address are discussed in the following subsections.

4.9.1 Maximum Transmission Units (MTU)

Most networks are limited in the number of bytes that can be physically transmitted in a single transaction. Each NI therefore has an associated *maximum transmission unit* (MTU), which is the maximum packet size that can be sent or received. If the size of a packet exceeds the network's MTU, the IP layer fragments the packet for transmission. Similarly, the IP layer on the receiving node reassembles the fragments into the original packet.

The minimum MTU allowed by pNA+ is 64 bytes. There is no maximum limit. A larger MTU leads to less fragmentation of packets, but usually increases the internal memory requirements of the NI. Generally, an MTU between 512 bytes and 2K bytes is reasonable. For example, the MTU for Ethernet is 1500.

4.9.2 Hardware Addresses

In addition to its Internet address, every NI has a *hardware address*. The Internet address is used by the IP layer, while the hardware address is used by the network driver when physically transferring packets on the network. The process by which Internet addresses are mapped to hardware addresses is called address resolution and is discussed in section 4.10.

Unlike an Internet address, which is four bytes long, the length of a hardware address varies depending on the type of network. For example, an Ethernet address is 6 bytes while a shared memory address is usually 4 bytes. pNA+ can support hardware addresses up to 14 bytes in length. The length of a NI's hardware address must be specified.

4.9.3 Control Flags

Each NI has a set of flags that define optional capabilities, as follows:

ARP This is used to enable or disable address resolution (see section 4.10).

BROADCAST This is used to tell pNA+ if the NI supports broadcasting. If you attempt to broadcast a packet on a network with this flag disabled, pNA+ returns an error.

EXTLOOPBCK	If this is disabled, pNA+ "loops back" packets addressed to itself. That is, if you send a packet to yourself, pNA+ does not call the NI, but the packet is processed as if it were received externally. If this flag is enabled, pNA+ calls the NI.
POLL	If this is set, the **ni_poll** service is called by the pSOS+ daemon task **pNAD**. This flag is normally used in conjunction with pROBE+ and XRAY+.
POINTTOPOINT	If this is set, the NI is a point-to-point interface.
RAWMEM	If this is set, pNA+ passes packets in the form of *mblk* (message block) linked lists (see Section 4.11, "Memory Management.") Similarly, the driver announces packets by passing a pointer to the message block.
UP	If this is set, the intial mode of the NI is up.

Note that if the **ARP** flag is enabled, the **BROADCAST** flag must also be set (see section 4.10).

4.9.4 Network Subnet Mask

A network can have a network mask associated with it to support subnet addressing. The network mask is a 32-bit value with ones in all bit positions that are to be interpreted as the network portion. See section 4.3.2 for a discussion on subnet addressing.

4.9.5 Destination Address

In point-to-point networks, two hosts are joined on opposite ends of a network interface. The destination address of the companion host is specified in the pNA+ NI Table entry **DSTIPADDR** for point-to-point networks.

4.9.6 The NI Table

pNA+ stores the parameters described above for each NI in the *NI Table*. The size of the NI Table is determined by the pNA+ Configuration Table entry **NC_NNI**, which defines the maximum number of networks that can be connected to pNA+.

Entries can be added to the NI Table in one of two ways:

1. The pNA+ Configuration Table entry **NC_INI** contains a pointer to an Initial NI Table. The contents of the Initial NI Table is copied to the actual NI Table during pNA+ initialization.
2. The pNA+ system call **add_ni()** can be used to add an entry to the NI Table dynamically, after pNA+ has been initialized.

4.10 Address Resolution and ARP

Every NI has two addresses associated with it -- an Internet address and a hardware address. The IP layer uses the Internet address, while the network driver uses the hardware address. The process by which an Internet address is mapped to a hardware address is called *address resolution.*

In many systems, address resolution is performed by the network driver. The address resolution process, however, can be difficult to implement. Therefore, to simplify the design of network drivers, pNA+ provides the capability of resolving addresses internally. To provide maximum flexibility, this feature can be optionally turned on or off, so that, if necessary, address resolution can still be handled at the driver level.

pNA+ goes through the following steps when performing address resolution:

1. pNA+ examines the NI flags (see section 4.9.3) to determine if it should handle address resolution internally. If not (i.e. the **ARP** flag is disabled), pNA+ simply passes the Internet address to the network driver.
2. If the **ARP** flag is enabled, pNA+ searches its ARP Table (see section 4.10.1) for an entry containing the Internet address. If an entry is found, the corresponding hardware address is passed to the NI.
3. If the Internet address is not found in the ARP Table, pNA+ uses the Address Resolution Protocol (see section 4.10.2) to obtain the hardware address dynamically.

4.10.1 The ARP Table

pNA+ maintains a table called the *ARP Table* for obtaining a hardware address, given an Internet address. This table consists of <Internet address, hardware address> tuples.

The ARP Table is created during pNA+ initialization; the pNA+ Configuration Table entry **NC_NARP** specifies its size. Entries can be added to the ARP Table in one of three ways:

1. An Initial ARP Table can be supplied. The pNA+ Configuration Table entry **NC_IARP** contains a pointer to an Initial ARP Table. The contents of the Initial ARP Table are copied to the actual ARP Table during pNA+ initialization.
2. Internet-to-hardware address associations can be determined dynamically by the ARP protocol. When pNA+ uses ARP to dynamically determine an Internet-to-hardware address mapping, it stores the new <Internet address, hardware address> tuple in the ARP Table. This is the normal way that the ARP Table is updated. The next section explains how ARP operates.
3. ARP Table entries can be added dynamically by using **ioctl()**.

4.10.2 Address Resolution Protocol (ARP)

pNA+ uses the *Address Resolution Protocol (ARP)* to determine the hardware address of a node dynamically, given its Internet address. ARP operates as follows:

1. A sender, wishing to learn the hardware address of a destination node, prepares and broadcasts an ARP packet containing the destination Internet address.
2. Every node on the network receives the packet and compares its own Internet address to the address specified in the broadcasted packet.
3. If a receiving node has a matching Internet address, it prepares and transmits to the sending node an ARP reply packet containing its hardware address.

ARP can be used only if all nodes on the network support it. If your network consists only of pNA+ nodes, this requirement is of course satisfied. Otherwise, you must make sure that the non-pNA+ nodes support ARP. ARP was originally developed for Ethernet networks and is usually supported by Ethernet drivers. Networks based on other media might or might not support ARP.

pNA+ treats Internet packets differently than ARP packets. When pNA+ calls an NI, it provides a *packet type* parameter, which is either **IP** or **ARP**. Similarly, when pNA+ receives a packet, the NI must also return a packet type. All network drivers that support ARP must have some mechanism for attaching this packet type to the packet. For example, Ethernet packets contain **type** fields. For NIs that do not support ARP, the packet type parameter can be ignored on transmission, and simply set to **IP** for incoming packets.

4.11 Memory Management

As packets move across various protocol layers in pNA+ they are subject to several data manipulations, including

- Addition of protocol headers
- Deletion of protocol headers
- Fragmentation of packets
- Reassembly of packets
- Copying of packets

pNA+ is designed with specialized memory management so that such manipulations can be done optimally and easily.

pNA+ allows configuration of its memory management data structures via the pNA+ Configuration Table. These structures are critical to its performance; hence, understanding the basics of pNA+ memory management is crucial to configuring your system optimally.

The basic unit of data used internally by pNA+ is called a *message*. Messages are stored in message structures. A message structure contains one or more message block triplets, linked via a singly-linked list. Each message block triplet contains a contiguous block of memory defining part of a message. A complete message is formed by linking such message block triplets in a singly-linked list.

Each message block triplet contains a Message Block, a Data Block, and a Buffer. Figure 9 illustrates the message block triplet.

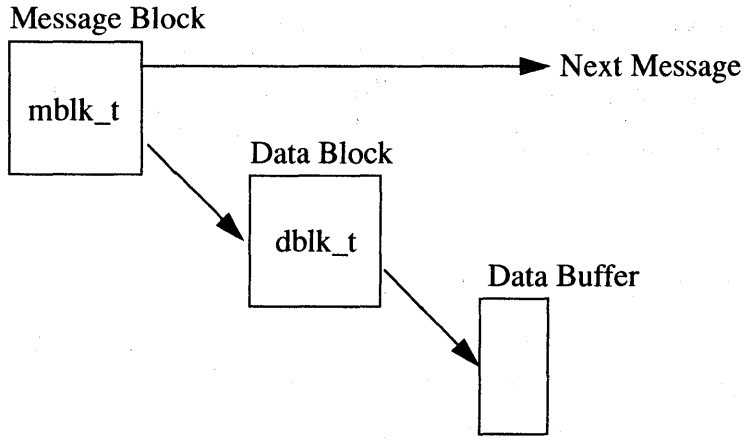

Figure 9. Message Block Triplet

A *message block* contains the characteristics of the partial message defined by the message block triplet. A *data block* contains the characteristics of the buffer to which it points. A *buffer* is a contiguous block of memory containing data.

A data block may be contained in several message block triplets. However, there is a one-to-one correspondence between data blocks and buffers. The C language definitions of the data structures, for message blocks and data blocks, are in the header file **<pna.h>**.

Figure 10 illustrates a complete message formed by a linked list of message block triplets.

The basic unit of transmission used by protocol layers in pNA+ is a *packet*. A packet contains a protocol header and the data it encapsulates. Each protocol layer tags a header to the packet and passes it to the lower layer for transmission. The lower layer in turn uses the packet as encapsulated data and tags its protocol header and passes it to its lower layer. Packets are stored in the form of messages.

The buffers in pNA+ are used to store data, protocol headers, and addresses. Data is passed into pNA+ via two interfaces. At the user level, data is passed via the **send()**, **sendto()** and **sendmsg()** service calls. At the NI interface, data is passed via the "Announce Packet" call (See Section 4.9, "Network Interfaces").

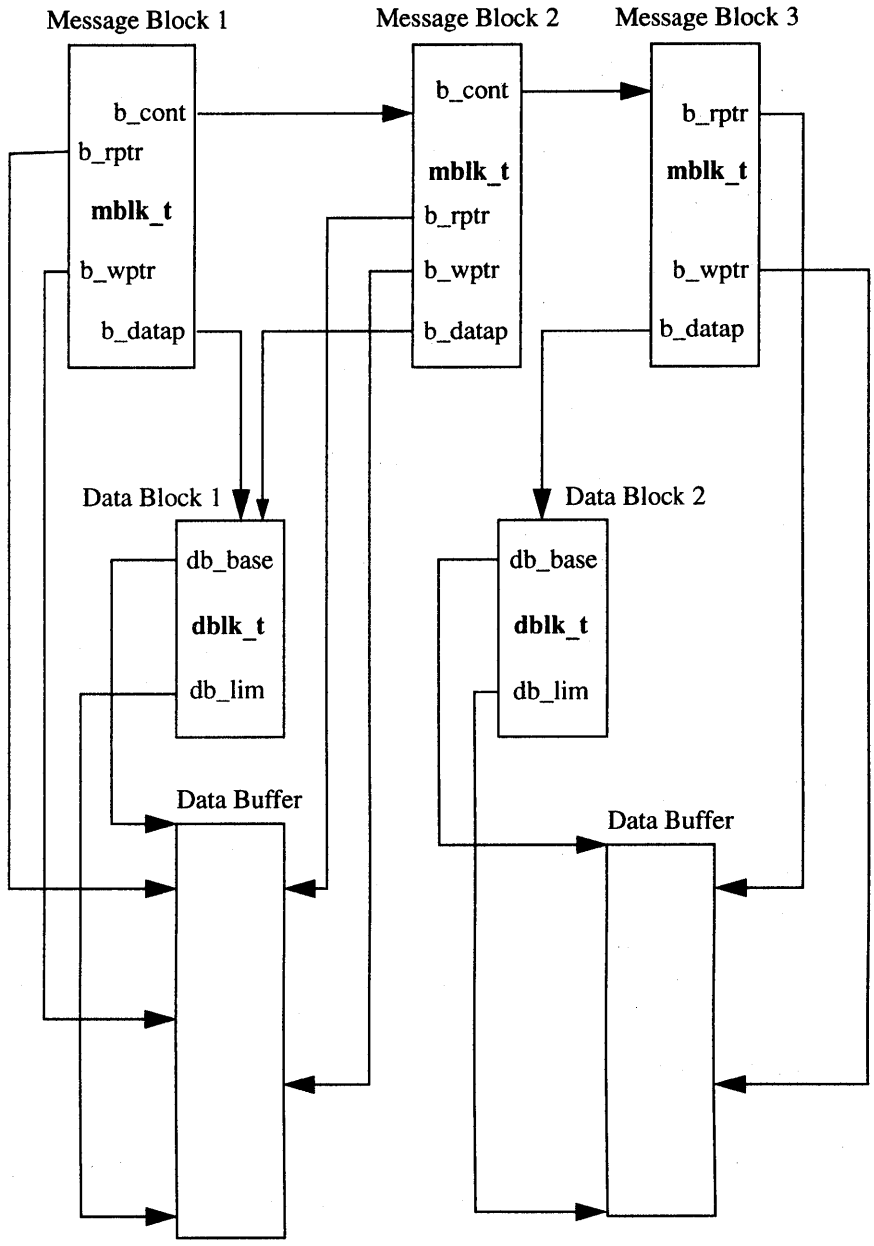

Figure 10. Message Block Linkage

pNA+ allocates a message block triplet and copies data from the external buffer to the buffer associated with the triplet. The message is then passed to the protocol layers for further manipulation. As the data passes through various protocol layers, additional message block triplets are allocated to store the protocol headers and are linked to the message. pNA+ also allocates temporary message block triplets to store socket addresses during pNA+ service calls.

As the messages pass through the protocol layers, they are subjected to various data manipulations (copying, fragmentation, and reassembly). For instance, when preparing a packet for transmission, the TCP layer makes a copy of the packet from the socket buffer, tags a TCP header, and passes the packet to the IP layer. Similarly, the IP layer fragments packets it receives from the transport layer (TCP, UDP) to fit the MTU of the outgoing Network Interface.

pNA+'s memory management is optimized to perform such operations efficiently and maximize performance by avoiding physical copying of data. For instance, copying of message block triplets is achieved by allocating a new message block, associating it with the original data block, and increasing the reference count to the original data block. This avoids costly data copy operations.

4.12 Memory Configuration

During the initialization of pNA+, various memory structures are created and initialized. The initialization sequence creates message blocks, data blocks, and data buffers of multiple sizes. The number of each is configurable in the pNA+ Configuration Table. pNA+ provides entries in the configuration table to specify the number of message blocks and data buffers. Since there is a one-to-one relationship between data blocks and data buffers, pNA+ allocates a data block for every buffer configured in the system.

The memory configuration of pNA+ is critical to its performance. Configuring too few buffers or wrong sizes leads to reduced performance. Configuring too many buffers wastes memory.

Optimal performance can be achieved empirically by tuning the following configurable elements:

- Number of message blocks
- Buffer configuration
- MTU-size buffers
- 128-byte buffers
- Zero-size buffers

The following sections give general configuration guidelines.

4.12.1 Buffer Configuration

Buffer configuration is specified via the **nc_bcfg** element in the pNA+ Configuration Table (See the *pSOSystem Programmer's Reference*). It allows you to configure application-specific buffer sizes into the system. There are two attributes associated with a buffer configuration -- buffer size and the number of buffers.

pNA+ copies data into its internal buffers via two interfaces. It copies data from the user buffers to its internal buffers during **send()**, **sendto()**, and **sendmsg()** service calls. It copies data from the NI buffers to its internal buffers during "Announce Packet" calls.

pNA+ allows buffers of multiple sizes to be configured into the system. In order to allocate a buffer to copy data, it first selects the buffer size, using the following best-fit algorithm:

1. pNA+ first tries to find an exact match for the data buffer.
2. If there is no such buffer size available, pNA+ searches for the smallest sized buffer that can contain the requested size.
3. If there is none, pNA+ selects the maximum buffer size configured.

Once a size is selected, pNA+ checks for a free buffer from the selected size's buffer list. If none are available, pNA+ blocks the caller on a blocking call, or returns null on a non-blocking call. If the size of the buffer is not sufficient to copy all of the data, pNA+ copies the data into multiple buffers.

For optimal configuration, pNA+ should always find an exact match when doing buffer size selection. Thus, the configuration should have buffer

sizes equal to the MTU of the NI's configured in pNA+ to satisfy the requirement at the NI interface, and buffer sizes equal to the user buffer sizes specified in the **send()**, **sendto()**, and **sendmsg()** service calls to satisfy user interface requirements. The number of buffers to be configured for each size depends on the socket buffer size and incoming network traffic.

pNA+'s flexible memory configuration provides multiple buffer sizes. However, 128-byte and zero-size buffers have special meanings. 128-byte buffers are used internally by pNA+ for storing protocol headers and for temporary usage. These buffers must always be configured for pNA+ to function. Zero-size buffers are used to create message block triplets with externally specified data buffers (See Section 4.13, "Zero Copy Options," and **pna_esballoc()** in the *pSOSystem Programmer's Reference*).

4.12.1.1 MTU-Size Buffers

When a non-zero copy NI is configured in pNA+, data is copied from the NI buffers to pNA+ internal buffers. Hence, it is optimal to have MTU-size buffers configured in the system. The number of buffers that should be configured depends on the incoming network traffic on that NI.

4.12.1.2 Service-Call-Size Buffers

Data is copied from user buffers to pNA+ internal data buffers during **send()**, **sendto()**, and **sendmsg()** service calls. For optimal performance, pNA+ should be configured with buffer sizes specified in the service calls. The optimal number of buffers depends on the buffer size of the socket.

4.12.1.3 128-Byte Buffers

pNA+ uses 128-byte buffers to store protocol headers and addresses. The number of protocol headers allocated at any given time depends on the number of packets sent or received simultaneously by the protocol layers in pNA+. The number of packets sent or received by pNA+ varies with the number of active sockets and with socket buffer size. The number of packets that can exist per active socket is the socket buffer size divided by the MTU of the outgoing NI. pNA+ service calls also use 128-byte buffers for temporary purposes; they use a maximum of three buffers per call.

4.12.1.4 Zero-Size Buffers

Zero-size buffers are used during **pna_esballoc** service calls to attach externally supplied user buffers to a message block and a data block. When zero-size buffers are specified, pNA+ allocates only a data block; that is, the associated buffer is not allocated.

The optimal number of zero-size buffers to be configured depends on the number of externally specified buffers that can attached to pNA+ message blocks; that is, the number of times **pna_esballoc** is used. (For more details, see Section 4.13, "Zero Copy Options.")

4.12.2 Message Blocks

The memory manager in pNA+ is highly optimized for data copy and fragmentation. During these operations, pNA+ allocates an additional message block and reuses the original data block and buffer. The number of copy or fragmentation operations done by pNA+ per buffer depends on the size of the buffer and on the MTU size of the NI's configured in the system.

The maximum number of fragments for buffers of sizes less than the smallest MTU is two, and the maximum number of fragments for all other buffers is the buffer size divided by the MTU.

The number of message blocks configured in the system should equal the total number of fragments that can be formed from the buffers configured in the system. In most cases, it is sufficient to configure the total number of message blocks to be twice the total number of buffers configured in the system.

4.12.3 Tuning pNA+

pNA+ also provides statistics for buffer and message block usage via the **ioctl()** service call. The SIOCGDBSTAT command can be used to return buffer usage, and SIOCGMBSTAT can be used to get message block usage.

These commands provide information on the number of times tasks waited for a buffer, the number of times a buffer was unavailable, the number of free buffers, and the total number of buffers configured in the system. You can use this information to tweak the message block and data buffer configuration.

4.13 Zero Copy Options

Copying data is an expensive operation in any networking system. Hence, eliminating it is critical to optimal performance. pNA+ performs data copy at its two interfaces. It copies data from the user buffer to pNA+ internal buffers during **send()**, **sendto()**, and **sendmsg()** service calls, and vice versa during **recv()**, **recvfrom()**, and **recvmsg()** calls. A data copy is performed between the NI and pNA+ buffers when data is exchanged.

Because the pNA+ memory manager is highly optimized to eliminate data copy, data is copied only at the interfaces during data transfers. In order to maximize performance, pNA+ provides options to eliminate data copy at its interfaces, as well. These options are referred to as "zero copy" operations. pNA+ extends the standard Berkeley socket interface at the user level and provides an option at the NI level to support zero copy operations.

Zero copy is achieved in pNA+ by providing a means of exchanging data at interfaces via message block triplets and by enabling access to its memory management. The zero copy operations provided at the interfaces are independent of each other; that is, an application can choose either one, or both. In most cases, the NI interface is optimized to perform zero copy, while retaining the standard interface at the socket level.

4.13.1 Socket Extensions

The **sendto()**, **send()**, **recv()**, and **recvfrom()** service calls are extended to support the zero copy option. An option is provided in the calls allowing data to be exchanged via message block triplets. An additional flag (MSG_RAWMEM) is provided in these service calls. When the **flags** parameter in these service calls is set to MSG_RAWMEM, the **buf** parameter contains a pointer to a message block triplet. (See the *pSOSystem Programmer's Reference.*)

When the zero copy option is not used, a buffer always remains in the control of its owner. For example, during a **send()** call, the address of the buffer containing data to be sent is passed to pNA+. As soon as the call returns, the buffer can be reused or de-allocated by its owner. pNA+ has copied the data into its internal buffers.

When the zero copy option is used, control of the buffer triplet passes to pNA+. When pNA+ finishes using the message block triplet, the triplet is freed. Similarly, on a **recv()** call, control of the buffer passes to the application, which is responsible for freeing the message block triplet.

Four service calls are provided to access pNA+ memory management. They are as follows:

pna_allocb() allocates a message block triplet that contains a data buffer of the size passed in as a parameter. The data buffer is internal to pNA+.

pna_freeb() frees a single message block triplet.

pna_freemsg() frees a message.

pna_esballoc() associates a message block and a data block with an externally specified buffer. **pna_esballoc()** returns a pointer to a message block triplet that contains a message block and a data block allocated by pNA+. The data buffer in the triplet is passed in as a parameter to the call.

4.13.2 Network Interface Option

The pNA+ network interface definition supports data exchange between pNA+ and an NI via message block triplets. If the RAWMEM flag is set in the NI flags, it indicates that the interface supports the zero copy operation, and the exchange of data between NI and pNA+ is in the form of message block triplets.

The pointers to the **pna_allocb()**, **pna_freeb()**, **pna_freemsg()**, and **pna_esballoc()** functions are passed to the NI driver during its **ni_init** function call. (See Section 4.9, "Network Interfaces.") These functions are used by the NI to gain access to pNA+ memory management routines.

4.14 Internet Control Message Protocol (ICMP)

ICMP is a control and error message protocol for IP. It is layered above IP for input and output, but it is really part of IP. ICMP can be accessed through the raw socket facility. pNA+ processes and generates ICMP messages in response to ICMP messages it receives.

ICMP can be used to determine if pNA+ is accessible on a network. For example, some workstations (such as SUN) provide a utility program called **ping**, which generates ICMP echo requests and then waits for corresponding replies and displays them when received. pNA+ responds to these ICMP messages sent by **ping**.

ICMP supports 7 unique message types, with each reserved to designate specific IP packet or network status characteristics, as follows:

TYPE	DESCRIPTION
1	ECHO REQUEST AND REPLY. This type is used to test/verify that the destination is reachable and responding. The **ping** utility relies on this ICMP message type.
2	DESTINATION UNREACHABLE. This message type is generated when an IP datagram cannot be delivered by a node. This type is further delineated by ancilliary *codes* defined as follows:

CODE	EXPLANATION
0	Network unreachable.
1	Host unreachable.
2	Protocol unreachable.
3	Port unreachable.
4	Fragmentation needed and DF is set.
5	Source route failed.
6	Destination network unknown.
7	Destination host unknown.
8	Source isolated.
9	Communication with destination network is administratively prohibited.
10	Communication with destination host is administratively unknown.
11	Network unreachable for type of service.
12	Host unreachable for type of service.

3	SOURCE QUENCH. This type is generated when buffers are exhausted at an intermediary gateway or end-host.
4	REDIRECT. This type is generated for a change of route.

TYPE	DESCRIPTION
5	TIME EXCEEDED FOR DATAGRAM. This type is generated when the datagram's time to live field has exceeded its limit.
6	TIMESTAMP REQUEST AND REPLY. This type is generated to request a timestamp.
7	ADDRESS MASK REQUEST AND REPLY. This type is sent to obtain a subnet address mask.

4.15 NFS Support

pNA+ can be used in conjunction with pHILE+ and pRPC+ to offer NFS support. To support NFS, pNA+ allows you to assign a hostname to your pNA+ system, and a user ID and group ID to each task. The hostname and user and group IDs are used when accessing NFS servers. Every task that uses NFS services must have a user ID and a group ID. These values are used by an NFS server to recognize a client task and grant or deny services based on its identity. Refer to your host system (NFS server) documentation for a further discussion of NFS protection mechanisms.

The pNA+ Configuration Table entry **NC_HOSTNAME** is used to define the hostname. This entry points to a null terminated string of up to 32 characters, which contains the hostname for the node.

The pNA+ Configuration Table entries **NC_DEFUID** and **NC_DEFGID** can be used to define default values for a task's user ID and group ID, respectively. Subsequent to task creation, the system calls **set_id()** and **get_id()** can be used to change or examine a task's user and group ID. Note that similar system calls [**setid_u()** and **getid_u()**] are provided by pHILE+. We recommend, however, that you use the **set_id()** and **get_id()** system calls provided in pNA+ for future compatibility.

4.16 MIB-II Support

pNA+ supports a TCP/IP Management Information Base, commonly known as MIB-II, as defined in the Internet standard RFC 1213. pSOSystem's optional SNMP (Simple Network Management Protocol) package uses this MIB-II to provide complete turn-key SNMP agent functionality.

pNA+'s MIB-II can also be accessed directly by application developers who have their own unique requirements. This section describes how this MIB can be accessed.

4.16.1 Background

RFC 1213 groups MIB-II objects into the following categories:

- System
- Interfaces
- Address Translation
- IP
- ICMP
- TCP
- UDP
- EGP
- Transmission
- SNMP

pNA+ contains built-in support for the IP, ICMP, TCP, and UDP groups. The Interfaces group is supported by pNA+ NIs. pSOSystem's SNMP library provides support for the System and SNMP groups. The Address Translation group is being phased out of the MIB-II specification. Its functionality is provided via the IP group. The Transmission group is not yet defined, and pNA+ does not include EGP, so neither of these groups are supported.

MIB-II objects, regardless of which category they fall into, can be classified as simple variables or tables. Simple variables are types such as integers or character strings. In general, pNA+ maintains one instance of each simple variable. For example, **ipInReceives** is a MIB-II object used to keep track of the number of datagrams received.

Tables correspond to one-dimensional arrays. Each element in an array (that is, each entry in a table) has multiple fields. For example, MIB-I includes an IP Route Table where each entry in the table consists of the following fields: **ipAdEntAddr**, **ipAdEntIfIndex**, **ipAdEntNetMask**, **ipAdEntBcastAddr**, **ipAdEntReasmMaxSize**.

4.16.2 Accessing Simple Variables

All MIB-II objects, regardless of type, are accessed by using the pNA+ **ioctl(int s, int command, int *arg)** system call. The parameter **s** can be any valid socket descriptor.

The **command** argument specifies a MIB-II object and the operation to be performed on that object. Per the SNMP standard, two operations are allowed. You can set the value of a MIB-II object (Set command) or retrieve an object's value (Get command). A valid **command** parameter is an uppercase string equal to the name of a MIB-II object prepended by either **SIOCG** or **SIOCS** for Get and Set operations, respectively. A complete list of permissable commands is provided in the *pSOSystem Programmer's Reference.*

The way **ioctl()** is used differs, depending on whether you are accessing simple variables or tables. For simple variables, **arg** is a pointer to a variable used either to input a value (for Set operations) or receive a value (for Get operations). **arg** must be typecast based on the MIB-II object type.

The following table shows the C language types used by pNA+ to represent different types of MIB-II objects.

MIB-II Object Type	pNA+ Representation
INTEGER	**long**
OBJECT IDENTIFIER	**char** * (as an ASCII string)
IpAddress	**struct in_addr** (defined in **pna.h**)
Counter	**unsigned long**
Gauge	**unsigned long**
TimeTicks	**unsigned long**
DisplayString	**char** *
PhysAddress	**struct sockaddr** (defined in **pna.h**)

The following code fragments demonstrate how to set and get the objects **ipInReceives**, and **ipForwarding**, respectively:

```
{
/* Get the value of ipInReceives */
  long s;
  unsigned long ip_input_pkts;

  /* socket type in following call is irrelevant */
  s = socket(AF_INET, SOCK_STREAM, 0);
  ioctl(s, SIOCGIPINRECEIVES, &ip_input_pkts);
  close(s);
  printf("%lu IP datagrams recvd\n", ip_input_pkts);
}


/* Set the value of ipForwarding */
int s; /* already open socket descriptor */
{
  long forwarding;
  /* get current status first */

  ioctl(s, SIOCGIPFORWARDING, &forwarding);
  if (forwarding == 1) puts("Forwarding was on");
  else /* forwarding == 2 */ puts("Forwarding was off");
  forwarding = 2; /* corresponds to not-forwarding */
  ioctl(s, SIOCSIPFORWARDING, &forwarding);
  puts("Forwarding turned off");
}
```

4.16.3 Accessing Tables

Accessing information stored in tables is more complicated than accessing simple variables. The complexity is primarily due to the SNMP specification and the fact that table sizes vary over time, based on the state of your system.

pNA+ defines C data structures for each MIB-II table. These definitions are contained in **<pna_mib.h>** and are shown in section 4.16.4. A table usually consists of multiple instances of the entries shown. pNA+ allows you to access any field in any entry, add table entries, and delete entries.

The key to understanding how to manipulate tables is to recognize that MIB-II table entries are not referenced by simple integers (like normal programming arrays). Rather one or more fields are defined to be index fields, and entries are identified by specifying values for the index fields. The index fields were selected so that they identify a unique table entry. The index fields are indicated in the MIB-II tables shown.

This raises the question, how do you know what are valid indices at any time. You obtain them with **ioctl()** the following way. First, declare a variable of type **mib_args** (this structure is defined in **<pna_mib.h>**) using the following syntax:

```
struct mib_args {
  long  len;/* bytes pointed to by buffer    */
  char  *buffer;/* ptr to table-specific struct array */
};
```

buffer points to an array of structures with a type corresponding to the table you want to access. **len** is the number of bytes reserved for **buffer**. The buffer should be large enough to hold the maximum possible size of the particular table being accessed.

Call **ioctl()** with **command** equal to the MIB-II object corresponding to the name of the table. **arg** is a pointer to the **mib_args** variable.

Upon return from **ioctl()**, the array pointed to by **arg** will have all of its index fields set with valid values. In addition, there will be one other field set with a valid value. This field is indicated as **default** in the tables shown.

After you obtain a list of indices, you may set or retrieve values from fields in the tables. You issue an **ioctl()** call with **command** corresponding to the name of a field and **arg** pointing to a table-specific data structure.

The following code fragment illustrates how all of this works by traversing the IP Route Table:

```
int s;  /* already opened socket descriptor */
{
 struct mib_iproutereq *routes;  /* the array of routes */
 struct mib_args arg;
 int num_routes, len, i;
```

```
    num_routes = 50;  /* default number of routes in array */
    routes = NULL;  /* to insure it is not free()d before it is allocated */

    /* loop until enough memory is allocated to hold all the routes */
    do {
      if (routes) {       /* if not the first iteration */
        free(routes);     /* free memory from the previous iteration */
        num_routes *= 2; /* allocate more space for the next try */
      }
      len = sizeof(struct mib_iproutereq) * num_routes; /* # of bytes */
      routes = (struct mib_iproutereq *)malloc(len); /* array itself */
      arg.len = len;
      arg.buffer = (char *)routes;
      ioctl(s, SIOCGIPROUTETABLE, (int *)&arg);
    }while (arg.len == len);  /* if full there may be more routes */

    num_routes = arg.len / sizeof(struct mib_iproutereq); /* actual # */
    puts("Destination   Next hop       Interface");
    for (i = 0; i < num_routes; i++) {  /* loop through all the routes */
      printf("0x%08X    0x%08X", routes[i].ir_idest.s_addr,
               routes[i].ir_nexthop.s_addr);
      ioctl(s, SIOCGIPROUTEIFINDEX, (int *)&routes[i]);
      printf("    %d\n", routes[i].ir_ifindex);
    }
    free(routes);
}
```

You can insert a new entry into a table by specifying an index field with a nonexistent value. The following code fragment shows an example of how to add an entry into the IP Route Table.

```
int s; /* already opened socket descriptor */
void add_route(struct in_addr destination,
                 struct in_addr gateway)
{
  struct mib_iproutereq route;

  route.ir_idest = destination;
  route.ir_nexthop = gateway;
  ioctl(s, SIOCSIPROUTENEXTHOP, &route);
}
```

You can delete a table entry by setting a designated field to a prescribed value. These fields and values are defined in RFC 1213. The following

code fragment provides an example of deleting a TCP connection from the TCP Connection Table so that the local port can be re-used:

```
int s;  /* already opened socket descriptor */

void delete_tcpcon(struct in_addr remote_addr, struct in_addr
                local_addr, short remote_port, short local_port)
{
  struct mib_tcpconnreq tcpconn;

  tcpconn.tc_localaddress = local_addr;
  tcpconn.tc_remaddress = rem_addr;
  tcpconn.tc_localport = local_port;
  tcpconn.tc_remport = rem_port;
  tcpconn.tc_state = TCPCS_DELETETCB;
  ioctl(s, SIOCSTCPCONNSTATE, &tcpconn);
}
```

4.16.4 MIB-II Tables

This section presents the MIB-II tables supported by pNA+ and their corresponding C language representations.

4.16.4.1 Interfaces Table

Structure and Elements	MIB-II Object	Type
struct mib_ifentry		
ie_iindex	ifIndex	index
ie_descr	ifDescr	
ie_type	ifType	default
ie_mtu	ifMtu	
ie_speed	ifSpeed	
ie_physaddress	ifPhysAddress	
ie_adminstatus	ifAdminStatus	
ie_operstatus	ifOperStatus	
ie_lastchange	ifLastChange	
ie_inoctets	ifInOctets	
ie_inucastpkts	ifInUcastPkts	
ie_nucastpkts	ifInNUcastPkts	
ie_indiscards	ifInDiscards	
ie_inerrors	ifInErrors	
ie_inunknownprotos	ifInUnknownProtos	

Structure and Elements	MIB-II Object	Type
ie_outoctets	ifOutOctets	
ie_outucastpkts	ifOutUCastPkts	
ie_outnucastpkts	ifOutNUcastPkts	
ie_outdiscards	ifOutDiscards	
ie_outerrors	ifOutErrors	
ie_outqlen	ifOutQLen	
ie_specific	ifSpecific	

4.16.4.2 IP Address Table

Structure and Elements	MIB-II Object	Type
struct mib_ipaddrreq		
ia_iaddr	ipAdEntAddr	index
ia_ifindex	ipAdEntIfIndex	default
ia_netmask	ipAdEntNetMask	
ia_bcastaddr	ipAdEntBcastAddr	
ia_reasmmaxsize	ipAdEntReasmMaxSize	

4.16.4.3 IP Route Table

Structure and Elements	MIB-II Object	Type
struct mib_iproutereq		
ir_idest	ipRouteDest	index
ir_ifindex	ipRouteIfIndex	
ir_nexthop	ipRouteNextHop	default
ir_type	ipRouteType	
ir_proto	ipRouteProto	
ir_mask	ipRouteMask	

4.16.4.4 IP Address Translation Table

Structure and Elements	MIB-II Object	Type
struct mib_ipnettomediareq		
inm_iifindex	ipNetToMediaIfIndex	index
inm_iaddr	ipNetToMediaNetAddress	index
inm_physaddress	ipNetToMediaPhysAddress	default
inm_type	ipNetToMediaType	

4.16.4.5 TCP Connection Table

Structure and Elements	MIB-II Object	Type
struct mib_tcpconnreq		
tc_localaddress	tcpConnLocalAddress	index
tc_localport	tcpConnLocalPort	index
tc_remaddress	tcpConnRemAddress	index
tc_remport	tcpConnRemPort	index
tc_state	tcpConnState	default

4.16.4.6 UDP Listener Table

Structure and Elements	MIB-II Object	Type
struct mib_udptabreq		
u_localaddress	udpLocalAddress	index
u_localport	udpLocalPort	index

4.16.5 SNMP Agents

The following IP group operations must be handled within an SNMP agent itself, rather than through **ioctl()**.

MIB-II Object	Operation	Comment
ipRouteIfIndex	Set	The value of this object cannot be set, since it is always determined by the IP address.
ipRouteMetric*	Both	An SNMP agent should return -1 as their value.
ipRouteAge	Get	An SNMP agent should return -1 as its value.
ipRouteMask	Set	The values of these objects can be interrogated but not changed.
ipRouteInfo	Get	An SNMP agent should return { 0 0 } as the value of this object.
ipRoutingDiscards	Get	An SNMP agent should return 0 as the value of this object.

4.16.6 Network Interfaces

Objects defined by the Interfaces group are maintained by the Network Interfaces configured in your system. These objects are accessed via the **ni_ioctl()** system call.

pNA+ uses **ni_ioctl()** when necessary to access Interfaces objects. **ni_ioctl()** is described in the *pSOSystem Programmer's Reference.*

4.17 Subcomponents

pNA+ can be "extended" by adding two *subcomponents*: pRPC+ and pX11+.

A pNA+ subcomponent is a block of code that extends the feature set of pNA+. Subcomponents are similar to all other components, with the caveat that they rely on pNA+ for resources and services.

pNA+ initializes each subcomponent after it completes its own initialization sequence. Like other components, subcomponents require RAM, which can be allocated from Region 0 or defined in a Configuration Table.

The pNA+ Configuration Table entry **NC_CFGTAB** points to a subcomponent table, which in turn contains pointers to Configuration Tables for all subcomponents.

pNA+ subcomponents share the pNA+ error code space for both fatal and nonfatal errors.

A pNA+ nonfatal error code has the form 0x50XX, where XX is the error value. A subcomponent error code is of the form 0x5NXX, where N is the subcomponent ID, and XX is the error value.

A pNA+ fatal error code has the form 0x5FXX, where XX is the fatal error value. A set of 32 fatal errors from the pNA+ fatal error space is allocated for each subcomponent beginning at 0x80 (see the *pSOSystem Programmer's Reference* manual for a complete listing of fatal and nonfatal pNA+ error codes).

4.17.1 pRPC+

pRPC+ is a pNA+ subcomponent; it provides a complete implementation of the Open Network Computing (ONC) Remote Procedure Call (RPC) and eXternal Data Representation (XDR) specifications. pRPC+ is designed to be source-code compatible with Sun Microsystem's RPC and XDR libraries. The following subsections describe those aspects of pRPC+ that are unique to the Integrated Systems implementation.

4.17.1.1 pRPC+ Architecture

pRPC+ depends on the services of other pSOSystem components in addition to pNA+. Figure 11 illustrates the relationship between pRPC+ and the other parts of pSOSystem.

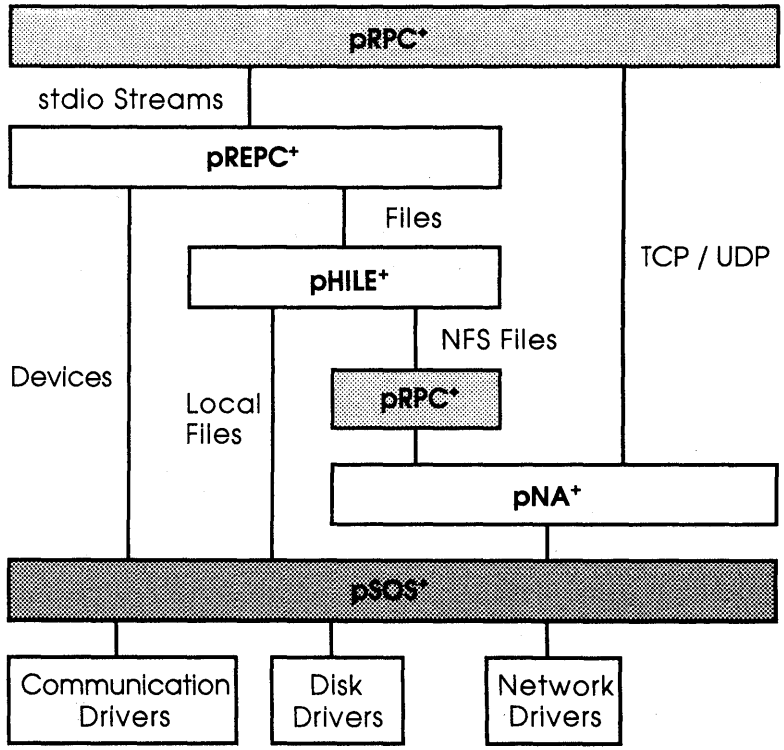

Figure 11. pRPC+ Dependencies

RPC packets use the TCP or UDP protocols for network transport. pNA+ provides the TCP/UDP network interface to pRPC+.

Direct access to XDR facilities, bypassing RPC, is supported by using memory buffers or stdio streams as a translation source or destination. I/O streams are managed by pREPC+. Streams may refer to pHILE+ managed files or directly to devices. pHILE+ accesses remote NFS files by using network RPCs, utilizing both pRPC+ and pNA+.

In addition to the communication paths shown on the diagram, pRPC+ also relies on pREPC+ for support of standard dynamic memory allocation. Consequently, XDR memory allocation within pRPC+ uses the same policy when insufficient memory is available as is used by applications that use the pREPC+ ANSI standard interface directly.

pRPC+ uses services provided directly by pREPC+ and PNA+. Installation of those components is prerequisite to the use of pRPC+. The installation of pHILE+ is only required if the ability to store XDR encoded data on local or remote disk files is desired.

pRPC+ must be installed in any system which will use pHILE+ for NFS, regardless of whether custom RPC/XDR code will be used or not. This is necessary because NFS is implemented using RPC/XDR. XDR is useful in conjunction with NFS to facilitate the sharing of raw data files between hosts which use different native representations of that data. Using XDR to write data files guarantees they can be correctly read by all hosts. NFS has no knowledge of file contents or structure so cannot perform any data translation itself.

4.17.1.2 Authentication

The RPC protocol allows the optional use of client authentication by RPC servers. When authentication is being employed, servers can identify the client task which made a specific request. Clients are identified by "credentials" included with each RPC request they make. Servers may refuse requests based upon the contents of the their credentials.

The representation of credentials is operating system specific because different operating systems identify tasks in different manners. Consequently, the RPC definition allows the use of custom credentials in addition to specifying a format for UNIX task credentials.

In order to facilitate the porting of UNIX clients to pSOSystem and interoperability between pSOSystem clients and UNIX servers, pRPC+ fully supports the generation of UNIX-style credentials.

The content of UNIX credentials are defined by the following data structure:

```
struct authunix_parms
{
    u_long      aup_time;      /* credential's creation time  */
    char        *aup_machname; /* hostname of client */
    int         aup_uid;       /* client's UNIX effective uid */
    int         aup_gid;       /* client's UNIX effective gid */
    u_int       aup_len        /* element length of aup_gids */
    int         *aup_gids;     /* array of groups user is in */
};
```

pRPC+ supports the standard RPC routines for manipulating UNIX compatible credentials: **authunix_create()** and **authunix_create_default()**. Both routines automatically set the value of the aup_time element. The **authunix_create()** routine takes as arguments the values of the remaining fields. The **authunix_create_default()** routine sets the values of the authunix_parms structure members from their pNA+ equivalents (the pNA+ configuration parameters are fully documented in the *pSOSystem Programmer's Reference* manual.

authunix_parms member	Value set by authunix_create_default()
aup_machine	pNA+ configuration parameter NC_HOSTNAME
aup_uid	pNA+ configuration parameter NC_DEFUID, may be changed on a per-task basis by the pNA+ call set_id().
aup_gid	pNA+ configuration parameter NC_DEFGID, may be changed on a per-task basis by the pNA+ call set_id().
aup_len, aup_gids	aup_len is always 0 so aup_gids is always empty.

4.17.1.3 Port Mapper

RPC supports the use of the networking protocols TCP and UDP for message transport. Because RPC and TCP/UDP use different task addressing schemes, clients must translate servers' RPC addresses to TCP/UDP addresses prior to making remote procedure calls. RPC uses a "port mapper" task running on each host which performs address translation for local servers. Prior to making a remote procedure call, clients contact the server's port mapper to determine the appropriate TCP/UDP destination address. (The port mapper protocol is handled within the RPC library and its existence and use are transparent to application programmers.)

At system initialization time pRPC+ automatically creates a port mapper task with the pSOS+ name **pmap**. The **pmap** task is started with a priority of 254. An application may change the priority of **pmap** via the standard pSOS+ service call **t_setpri()**.

4.17.1.4 Global Variable

pSOSystem tasks all run in the same address space. Consequently, global variables are accessible to and shared by every task running on the same processor. Whenever multiple tasks use the same global variable, they must synchronize access to it in order to prevent its value from being changed by one task while it is being used by another task. Synchronization can be achieved by using a mutex lock (implemented with a semaphore) or disabling task preemption around the regions of code which access the variable.

pRPC+ eliminates the need to use custom synchronization in RPC/XDR applications by replacing global variables with task-specific equivalents. Subroutines are provided in pRPC+ to provide access to the task-specific variables.

The following global variables are replaced by local variables in pRPC+:

Global Variable	Service Call	Description
svc_fdset	**get_fdset()**	Bit mask of used TCP/IP socket IDs
rpc_createerr	**rpc_getcreateerr()**	Reason for RPC client handle creation failure

Use of these pRPC+ subroutines is described in the *pSOSystem Programmer's Reference.*

4.17.2 pX11+

pX11+ is a pNA+ subcomponent that provides a complete implementation of MIT's X Windows Xlib client library. The pX11+ API is source code compatible with X Windows Version 11 Release 4.

pX11+ requires the services of pREPC+ as well as pNA+, and of course pSOS+. The following subsections explain the small number of features that differ from the standard MIT release.

4.17.2.1 Error Handling and exit( )

The asynchronous nature of X Windows complicates error handling. Requests issued prior to the emergence of the error condition cannot be "taken back." The approach normally taken when fatal errors arise is to print an error message and then exit. This is standard X Windows practice for UNIX environments. This approach is inconsistent with the pSOS+ environment.

Two types of error conditions can arise in pX11+: fatal and nonfatal run-time. Fatal errors force a pX11+ task to terminate, while a nonfatal run-time error produces an error condition which may be handled by the user. Fatal errors result from internal pX11+ failures (such as loss of display connection or state anomalies), and nonfatal run-time errors arise from client programming errors or server resource failures.

Each pX11+ client has two entry points for error reporting, one each for fatal and nonfatal run-time errors. The user may chose to implement a replacement for the default mechanisms.

If a pXlib call generates a fatal error condition, the default fatal error handler prints an error message using the pREPC+ standard error stream followed by an **exit()**, which is tailored to the pSOS+ environment. pX11+ implements **exit()** by first calling **pX_Shut()**, which returns all resources associated with the task, including task-specific memory, and then attempts a **t_delete()** to terminate the task. If **t_delete()** fails, **t_suspend()** is used to halt task execution.

The user's entry point for fatal error handling is set by the pXlib library call **XSetIOErrorHandler()**. A sample fatal error handler is:

```
int MyIOErrorHandler(display)
Display *display;
{
fprintf(stderr, "Fatal error!\n");
pX_Shut();
exit();          /* pREPC+ exit() */
}
```

In the case of nonfatal run-time errors, a separate entry point is provided that also uses the pREPC+ standard error stream. A nonfatal run-time error will output messages on this stream. The client task is not terminated unless a fatal error is encountered. The entry point for run-time error report is set by the pX11+ library call **XSetErrorHandler()**.

Both functions are described in Xlib Reference Manual, Vol. 2, O'Reilly and Associates, Sebastopol CA, 1990.

4.17.2.2 Environment Variables, Files, and Global Variables

An embedded computer system's user environment often lacks the amenities and features that are common to standard UNIX platforms. Among these are a programmable shell for creating resource alias names -- environment variables -- that are globally available to any process through system calls. Environment variables and methods for accessing their values and creating equivalences -- reproducing an environment -- for application processes, are handled differently in pX11+. The mechanism is tailored to the embedded nature of pSOS+.

pX11+ defines one resource type and it is used to create environment variables and their equivalences. The symbol **PX_RES_ENVIRONMENT** is defined for use by three pX11+-specific functions which manipulate environment variables and pX11+ resources in general.

The function **pX_Init()** is used to establish these equivalence pairs at initialization. **pX_SetArg()** and **pX_GetArg()** can be used to alter equivalences during run-time. The *pSOSystem Programmer's Reference* manual provides full descriptions of each function.

The user can set or adjust the pX11+ resource type via the following methods. A resource can be initialized by loading its value into the pX11+ module configuration table. During startup, a pX11+ client inherits the configuration table defaults. Secondly, the pX11+ client can arbitrarily modify the resource values through a programmatic interface.

The Xlib and X Toolkit libraries rely on environment variable resources to establish default and startup values. In the pX11+ domain, environment variable equivalences are set by constructing a catenated string containing equivalence pairs specified by type and value. pX11+ looks for the following environment variables: **HOME**, **DISPLAY**, **XAUTHORITY**, **RESOURCE_NAME**, **USER**, and **XENVIRONMENT**.

Certain functions supported by pX11+ require filenames to be used as arguments. If pHILE+ is configured into the pSOS+ environment, these references are mapped onto calls for file I/O. The pXlib code module handles this mapping, and returns appropriate error codes to the modules when no file system is available. The filenames must obey the pHILE+ conventions.

Applications can access files on pHILE+ filesystem partitions, MS-DOS disks, or across NFS using pHILE+ with pRPC+ and pNA+.

In pX11+, each client possesses one complete set of the global variables specific to X Windows. Each set is distinct from any other peer. The global variables are replicated for each pX11+ task. Each pX11+ task allocates 6110 bytes from RAM region 0 when created. This information should be used to size the data area for configuration purposes.

(Blank Page)

5 pHILE+ File System Manager

This chapter describes pSOSystem's file management option, the pHILE+ File System Manager. The following topics are discussed:

- Volume types
- How to mount and access volumes
- Conventions for files, directories, and pathnames
- Basic services for all volume types
- Blocking and deblocking
- Cache buffers
- Synchronization modes
- Organization of pHILE+ formatted volumes

5.1 Volume Types

From the point of view of pHILE+, a file system is a collection of volumes where each volume consists of a set of files. A volume can be a single device (such as a floppy disk), a partition within a device (such as a section of a hard disk), or even a remote device (such as an NFS server). What constitutes a volume is entirely up to you.

pHILE+ recognizes the following three types of volumes:

- **pHILE+ Formatted Volumes**

  These are devices that are formatted and managed using SCG proprietary data structures and algorithms that are optimized for real-time performance. pHILE+ formatted volumes offer high throughput, data locking, selectable cache write-through, and contiguous block allocation. pHILE+ formatted volumes can be a wide range of devices from floppy diskettes to write-once optical disks.

- **MS-DOS Volumes**

  These are devices that are formatted and managed according to MS-DOS conventions and specifications. MS-DOS volumes offer a convenient method for exchanging data between a pSOS+ system and a PC running the MS-DOS operating system. Because of its internal organization, an MS-DOS volume is much less efficient than a pHILE+ volume. In addition, MS-DOS volumes are suitable only for floppy diskettes. Hence, they should be used only when data interchange via floppy diskette is desired. pHILE+ currently supports five MS-DOS floppy disk formats. They are:

  - 360 Kbyte (5 1/4" double density)
  - 1.2 Mbyte (5 1/4" high density)
  - 720 Kbyte (3 1/2" double density)
  - 1.44 Mbyte (3 1/2" high density)
  - 2.88 Mbyte (3 1/2" high density).

- **NFS Volumes**

  NFS volumes allow you to access files on remote systems, such as Sun workstations, via the Network File System Protocol (NFS). Files located on an NFS server may be treated exactly as though they were on a local disk.

5.2 Working With Volumes

The following sections discuss how to access pHILE+ and MS-DOS volumes, what naming conventions are used, and volume formatting differences.

5.2.1 Mounting And Unmounting Volumes

Before a volume can be accessed, it must be mounted. The **mount_vol()** system call is used to mount pHILE+ formatted volumes, while the **pc_mount()** call is used to mount MS-DOS volumes, and **nfs_mount()** is used to mount NFS volumes.

pHILE+ maintains a mounted volume table, whose entries track and control mounted volumes in a system. The size of the mounted volume table, and hence the maximum number of volumes that can be mounted contemporaneously, is determined by the parameter **FC_NMOUNT** in the pHILE+ Configuration Table.

When a volume is no longer needed, it should be unmounted by using the **unmount_vol()** system call. When a volume is unmounted, its entry in the mounted volume table is removed.

Any task can unmount a volume. It does not have to be the same task that originally mounted the volume. A volume cannot be unmounted if it has any open files.

Detailed descriptions of **mount_vol()**, **pcmount_vol()**, **nfsmount_vol()**, and **unmount_vol()** are provided in the *pSOSystem Programmer's Reference* manual.

5.2.2 Naming Conventions and I/O

When a volume is mounted, the caller provides a 32-bit pSOS+ logical device number, which consists of a 16-bit major device number followed by a 16-bit minor device number. This logical device number serves as the volume's name while it is mounted. A volume name is given to pHILE+ as a string of two decimal numbers: one for the major device number and one for the minor number separated by a dot. "0.1" is an example of a volume name.

The interpretation of the device number by pHILE+ depends on the type of volume. For pHILE+ formatted volumes and MS-DOS volumes, the major device number identifies a user-supplied device driver associated

with the volume. When pHILE+ needs to read or write a volume, it makes a pSOS+ I/O system call specifying the volume's major device number. pSOS+ uses the major device number to find the device driver through its I/O Switch Table. The minor device number is simply passed to the driver. Refer to Chapter 7, "I/O System," for a discussion of pSOS+ I/O and pHILE+ drivers.

NFS volumes do not have device drivers per se. I/O requests directed to NFS volumes are routed through pRPC+ and pNA+ rather than standard pSOS+ I/O mechanisms. The volume name is used only to identify the volume while it is mounted.

5.2.3 MS-DOS and pHILE+ Formatted Volumes

Internally, pHILE+ treats MS-DOS and pHILE+ formatted volumes differently than NFS volumes. Each MS-DOS and pHILE+ formatted volume consists of a sequence of logical blocks and a file is a named collection of blocks.

In this model, a logical block is a device-independent addressable unit of storage. pHILE+ interacts with your device drivers in terms of logical blocks. On pHILE+ formatted volumes, the size of a logical block is defined by the pHILE+ Configuration Table entry **FC_LOGBSIZE**. This parameter has a large impact on system performance. Within limits, a larger logical block size will reduce data scattering on a device and improve throughput as a result of fewer I/O operations. On MS-DOS volumes, the logical block is fixed at 512 bytes.

Logical blocks are numbered starting with 0. On pHILE+ formatted volumes, a block address is a 32-bit entity, so a volume may contain up to 2^{32} blocks. The conversion between logical block numbers and physical storage units -- such as head, cylinder, and sector -- is handled by your device driver.

Before an MS-DOS or pHILE+ formatted volume can be used, it must be initialized. Initialization consists of two distinct operations: physical formatting and logical formatting.

When a device is physically formatted, markings are written on the storage medium (typically a magnetic surface) which delineate these basic storage units, usually sectors or blocks. After a device is formatted physically, it contains a set of readable and writable storage elements, but no data or other organizational information. Physically formatting a

device is purely a hardware operation and hence is deliberately left to you.

The logical format operation provides the organization for a volume. The logical organization refers to the way files are managed, space is allocated, and so on, and is determined by the volume type. The logical organization of pHILE+ volumes is explained in Section 5.8, "pHILE+ Formatted Volumes." Popular books and articles explain the logical organization of MS-DOS volumes.

The **init_vol()** system call is used to logically format a pHILE+ formatted volume while the **pcinit_vol()** system call is used to logically format an MS-DOS volume.

The **pcinit_vol()** system call should not be confused with the MS-DOS **FORMAT** command, which actually performs a logical and physical format. If you format a floppy disk or a hard disk on a PC using the **FORMAT** command, you do not need to use **pcinit_vol()**. To prepare a hard disk before mounting it as an MS-DOS volume, you *must* use the MS-DOS **FDISK** and **FORMAT** commands (or comparable utilities provided by some SCSI Controller Board vendors).

5.2.4 NFS Volumes

When used in conjunction with pRPC+ and pNA+, pHILE+ offers NFS (Network File System) client services. This means that pSOSystem nodes can access files on remote systems that support the NFS protocol (NFS servers) exactly as though they were on a local disk. The relationship is depicted in Figure 12.

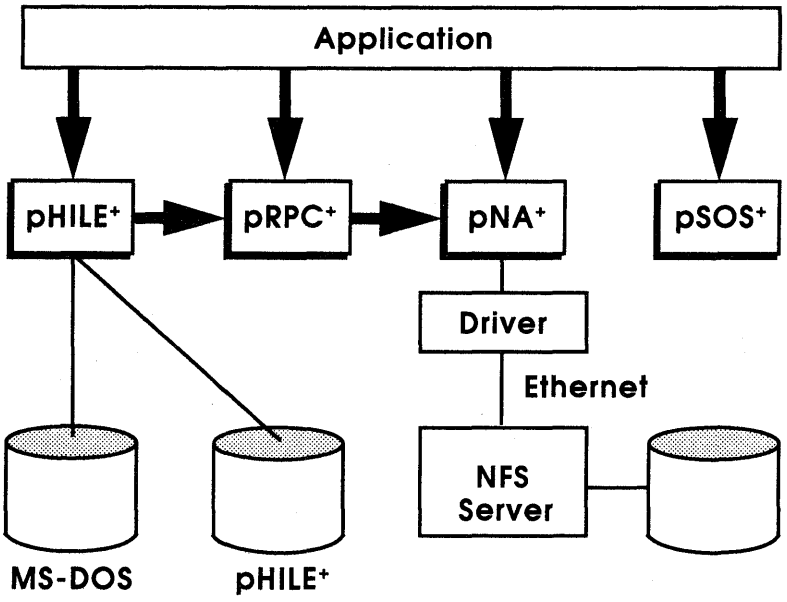

Figure 12. How Software Components Talk With NFS

To implement NFS, you must have these software elements:

- An application interface, to provide functions such as **open_f()** and **close_f()**. The application interface is provided by pHILE+.
- XDR services to put the data in a format that can be generally recognized, and Remote Procedure Calls to pass requests for NFS service to a server. pRPC+ provides RPC and XDR services.
- On the transport level, a socket interface that observes the User Datagram Protocol and the Internet Protocol, to carry the Remote Procedure Calls as UDP/IP messages for the server. pNA+ provides a UDP/IP transport for communication with a server.

For the most part, you treat remote and local files the same way. There are some differences, however, which you must understand when using NFS volumes.

When an NFS client (for example, pHILE+) requests services from an NFS server, it must identify itself by supplying a user ID, group ID, and

hostname. These items are used by the server to accept or reject client requests. How these parameters are used depends on the server.

The hostname is a string of up to 31 characters and must be supplied in the pNA+ Configuration Table. The user ID and group ID are 32-bit numbers. Default values for these quantities are supplied in the pNA+ Configuration Table. They may also be examined and set for individual tasks by using the pNA+ **getid_u()** and **setid_u()** system calls, respectively.

The **nfsmount_vol()** system call also has some unique features. When mounting an NFS volume, you must specify the IP address of an NFS server and the name of a directory on that server, which will act as the volume's root directory.

5.3 Files, Directories, and Pathnames

pHILE+ defines two types of files: ordinary files and directory files. An ordinary file contains user-managed data. A directory file contains information necessary for accessing ordinary and/or other (sub)directory files under this directory.

Every volume contains at least one directory file called the ROOT directory. From it can emanate a tree structure of directories and ordinary files to an arbitrary depth. Of course, the ROOT directory may contain only ordinary files, yielding a common, one-level structure.

Files may not cross over volumes and therefore cannot be larger than the volumes on which they reside. Every file is uniquely identified by using a pathname. A pathname specifies a path through a directory structure that terminates on a target file.

Pathnames are either absolute or relative. An absolute pathname always begins with a volume name and specifies a complete path through the directory tree leading to a file, starting at the volume's ROOT directory. A relative pathname identifies a file by specifying a path relative to a predefined directory, called the current directory. The current directory is unique for each task. It may be set and changed with the **change_dir()** system call.

An example of an absolute pathname is **0.3/fun/movies/tarzan**. This pathname identifies the file **tarzan** in the directory **movies**, which is in the directory **fun**, which in turn is in the root directory on volume 0.3.

An example of a relative pathname is **food/fruit/apples**. **apples** is a file in the directory **fruit**, which is in the directory **food**, which is a directory in the current directory.

/sports/baseball (note the leading slash) is another example of a relative pathname. In this case, the file **baseball** is in the directory **sports**, which is in the root directory on the volume defined by the current directory.

Rules for naming files and specifying pathnames vary depending on the type of volume. On all volumes, however, the names containing only a single or double dot (. and ..) are reserved. A single dot refers to the current directory, while .. refers to the parent of the current directory.

5.3.1 Naming Files on pHILE+ Formatted Volumes

On pHILE+ formatted volumes, a file is named by an ASCII string consisting of 1 to 12 characters. The characters can be either upper or lowercase letters, any of the digits 0 - 9, or any of the special characters . (period), _ (underscore), $ (dollar sign), or - (dash). A name must begin with a letter or a period. Names are case sensitive -- ABc and ABC represent different files.

When giving a pathname, the volume, directory, and filenames all are separated by either a forward (/) or backward (\) slash. The following examples show permissible pathnames for files located on pHILE+ formatted volumes:

"0.1/fruit/apples"
apples
./apples

5.3.2 Naming Files on MS-DOS Volumes

Files located on MS-DOS volumes are named according to standard MS-DOS naming conventions. Note the differences from the rules described above. First, MS-DOS filenames are not case sensitive (that is, abc and ABC name the same file). And second, MS-DOS names have two parts: a filename and an extension. The filename can be from one to eight characters and the extension may be from zero to 3 characters. Filenames and extensions are separated by a dot (.).

5.3.3 Naming Files on NFS Volumes

On NFS volumes, a file is named by sequence of up to 64 characters. All characters, except backslash (\) and zero are allowed. Filenames and directory names are separated in pathnames by forward slashes (/). If pHILE+ encounters a symbolic link while traversing an NFS pathname, it recursively expands the link up to three levels of nesting.

5.4 Basic Services for All Volumes

This section describes basic services that can be used with all types of volumes. For detailed descriptions of the system calls discussed in this section, see the *pSOSystem Programmer's Reference* manual.

5.4.1 Opening and Closing Files

Before a file can be read or written, it must be opened with the **open_f()** system call. **open_f()** accepts as input a pathname that specifies a file, and a mode parameter, which has meaning only when opening files located on NFS volumes. **open_f()** returns a small integer called a file ID (FID) that is used by all other system calls that reference the file.

A file may be opened by more than one task at the same time. Each time a file is opened, a new FID is returned.

When a file is opened for the first time, pHILE+ allocates a data structure for it in memory called a file control block (FCB). The FCB is used by pHILE+ to manage file operations and is initialized with system information retrieved from the volume on which the file resides.

All subsequent open calls on the file use the same FCB; it remains in use until the last connection to the file is closed. At that time, the FCB is reclaimed for reuse. The **close_f()** system call is used to terminate a connection to a file; it should be used whenever a file connection is no longer needed.

At pHILE+ startup, a fixed number of FCBs are created, reflecting the maximum number of permissible concurrently open files specified in the pHILE+ Configuration Table entry **FC_NFCB**.

In addition to the FCB, pHILE+ uses a system data structure called an open file table to manage open files. Every task has its own open file table, which is used by pHILE+ to store information about all of the files

that have been opened by that task. Each entry in an open file table controls one connection to a file. The FID mentioned above is actually used to index into a task's Open File Table.

The size of these open file tables is specified in the pHILE+ Configuration Table entry **FC_NCFILE**. This parameter sets a limit on the number of files which a task can have open at the same time.

Figure 13 shows the relationship between the system data structures discussed in this section.

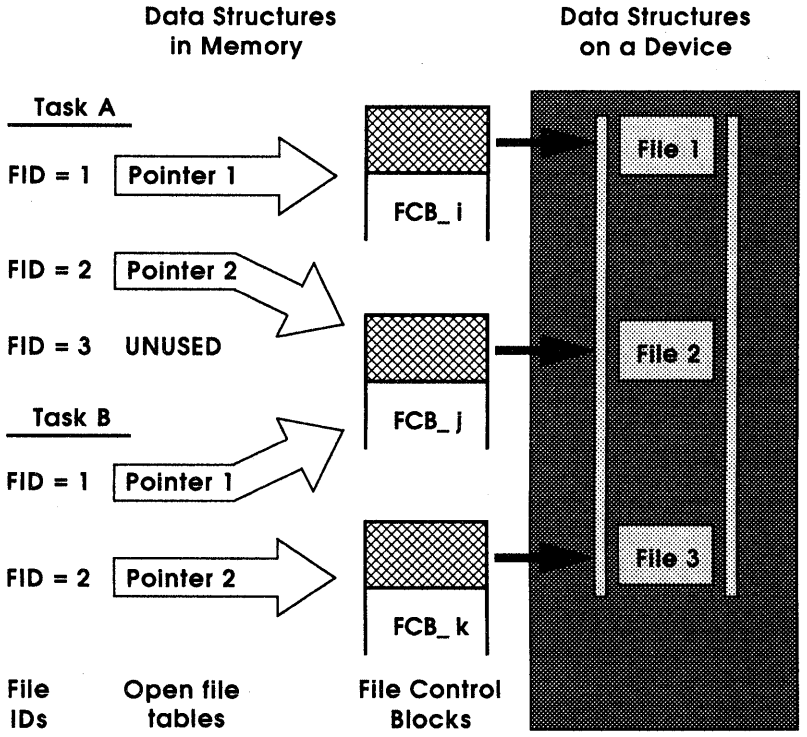

Figure 13. The Relationship Among a File ID, a File Control Block, and a File

5.4.2 Reading And Writing

Once a file is open, it may be read or written with the **read_f()** and **write_f()** system calls, respectively.

read_f() accepts as input an FID identifying the file to read, the address of a user data buffer to receive the data, and the number of bytes to read. Data transfer begins at the byte indicated by the position pointer, as explained in the next section.

read_f() returns the number of bytes transferred from the file to the user's buffer. If this value is less than the number requested and the return code does not indicate that an error occurred, then the end-of-file has been reached. Attempting to read beyond the end-of-file is not considered an error.

The **write_f()** system call is used to write data to a file. **write_f()** is similar to **read_f()**. It accepts as input an FID to identify a file, the address of a user data buffer containing data, and the number of bytes to transfer. Data transfer begins at the byte indicated by the position pointer, as explained in the next section. **write_f()** will always transfer the number of bytes requested, unless the target volume runs out of space or an error occurs.

5.4.3 Positioning Within Files

From the user's point of view, a file is a numbered sequence of bytes. For example, if a file contains 210 bytes, they are numbered 0 through 209.

For every connection established by **open_f()**, pHILE+ maintains a position pointer that marks the next byte to read or write. The position pointer is a 32-bit unsigned integer and is initialized to 0 by **open_f()**. Every read or write call advances the position pointer by the number of bytes transferred by that operation. In this way, a file can be read or written sequentially. The position pointer will be equal to the number of bytes in the file when the end-of-file is reached. In the example cited above, the position marker will be 210 after the last byte is read.

The **lseek_f()** system call can be used to relocate a position pointer. **lseek_f()** accepts three input parameters. The first parameter is an FID used to specify a file. The second parameter is an offset that specifies the number of bytes by which the position pointer should be "moved."

The third parameter specifies that the move should be relative to one of the following:

- The beginning of file
- The end of file
- The current position

pHILE+ does not allow positioning beyond the end of a file. Any attempt to do so results in an error code being returned. The position pointer is left unchanged.

5.4.4 Creating Files and Directories

Because of the differences between ordinary files and directory files, separate system calls are provided for creating files and directories. The **create_f()** system call is used to create an ordinary file. **make_dir()** is used to create directories. When an ordinary file is created, an entry for it is added to its parent directory. Both ordinary and directory files are initially empty.

When creating an ordinary file on a pHILE+ formatted volume, you must specify an expansion unit. This parameter controls the incremental growth of the file. Details on this parameter can be found in section 5.8.2.5.

5.4.5 Changing Directories

The current directory for a task can be set and altered using the **change_dir()** system call. **change_dir()** accepts as input a pathname specifying the new directory. This pathname can be either an absolute or relative pathname. Once the new directory is set, all subsequent relative pathnames are interpreted with respect to the new current directory.

pHILE+ does not assume a default current directory for any task. If a task intends to use relative pathnames, then it must call **change_dir()** at least once.

On pHILE+ formatted volumes, the current directory may be deleted. The results of using a relative pathname after the current directory has been deleted is unpredictable and should never be attempted.

5.4.6 Moving and Renaming Files

The **move_f()** system call allows a volume's directory tree structure to be modified by moving a file from one directory to another. On MS-DOS volumes, only ordinary files may be moved. On pHILE+ formatted volumes and NFS volumes, ordinary and directory files may be moved. When a directory is moved, all of the files and subdirectories are also moved.

move_f() can be used to rename a file by simply "moving" it within the same directory. Actually, **move_f()** is a misnomer, because **move_f()** never really moves data, it only manipulates directory entries.

Files may not be moved between volumes.

5.4.7 Deleting Files

Ordinary and directory files may be deleted (removed) by using the **remove_f()** system call. A file may not be removed if it is open or if it is a non-empty directory file.

5.5 Blocking/Deblocking

This and the following sections discuss some internal implementation issues that are relevant only for MS-DOS and pHILE+ formatted volumes. Understanding the material in these sections can help you improve the performance of your system.

From the user's point of view, a file is a sequence of bytes. Internally, however, pHILE+ implements a file as a sequence of logical blocks, and interacts with your driver in units of blocks. Therefore, for each user I/O request, pHILE+ must map the requested data bytes into logical blocks. On top of this, your device driver must, in turn, translate logical blocks into physical storage units. This process of translating bytes into blocks is called *blocking* and *deblocking*. The following scenarios illustrate how blocking and deblocking work.

When a **read_f()** operation requests bytes that are within a block, pHILE+ reads the entire block and then extracts the referenced bytes from it (deblocking).

When a **write_f()** operation writes bytes that are within a block, pHILE+ reads the entire block, merges the new data into it (blocking), and then writes the updated block back to the volume.

When a **read_f()** or **write_f()** operation references bytes that fit into an entire block or blocks, pHILE+ transfers the bytes as entire block(s). No blocking/deblocking is necessary.

When a **read_f()** or **write_f()** operation references bytes that straddle multiple blocks, the operation is broken down into separate actions. The bytes at the beginning and end of the sequence will require blocking/deblocking. The bytes that fill blocks in the middle of the sequence, if any, are transferred as entire blocks.

Note that read and write operations are most efficient if they start at block boundaries and have byte counts that are integral multiples of the block size, since no blocking/deblocking is required.

5.6 Cache Buffers

pHILE+ maintains a pool, or cache, of buffers for blocking/deblocking purposes. The number of cache buffers in your system is determined by the pHILE+ Configuration Table entry **FC_NBUF**. The size of the buffers in the buffer cache is determined by the pHILE+ Configuration Table entry **FC_LOGBSIZE**. Each buffer, when in use, holds an image of a logical block. A buffer can contain ordinary file data, directory file data, or system data structures. To improve system performance, pHILE+ uses the buffers as an in-memory cache for data recently retrieved from a device.

When pHILE+ needs to access a logical block, it first checks to see if an image of the block is contained in a cache buffer. If yes, pHILE+ simply works with the cache buffer in memory. There is no need for a physical I/O operation, thus improving performance.

Buffers in the cache are maintained using a least-recently-used algorithm. This means that if pHILE+ needs to use a buffer and they are all in use, then the buffer that has been untouched the longest, regardless of volume, is reused.

Before reusing a buffer, pHILE+ must test to see if the data in the buffer has been modified (e.g. because of a **write_f()** operation). If the data has been changed, then pHILE+ must call your driver to transfer the buffer's data to the volume before it can be reused. If the buffer has not been modified (for example, the data was only read), then the data on the volume is identical to that in the buffer, and the buffer can be reused.

It is worth noting that pHILE+ does not use the buffer cache under all conditions. If a read or write call involves all of the bytes within a block, then pHILE requests your driver to transfer the data directly between the volume and the user buffer specified in the system call. The buffer cache will be bypassed.

The following example illustrates how pHILE+ utilizes the buffer cache. pHILE+ receives a **write_f()** request for a sequence of bytes that covers 6 blocks, as follows (see Figure 14):

- The operation starts in middle of block 24, which is not in a cache. A buffer is obtained, and block 24 is copied into it. The respective bytes are written into the buffer.
- Blocks 25 and 26 are not in a cache. Since they are contiguous, a single physical write operation is used to write the bytes to blocks on the volume.
- Block 27 is in a cache buffer, so bytes are transferred to it, overwriting its old data.
- Block 28 is not in a cache, so a physical write operation is used to write the bytes to the block on the volume.
- Block 29 is in a cache buffer, so the respective bytes are written into it.

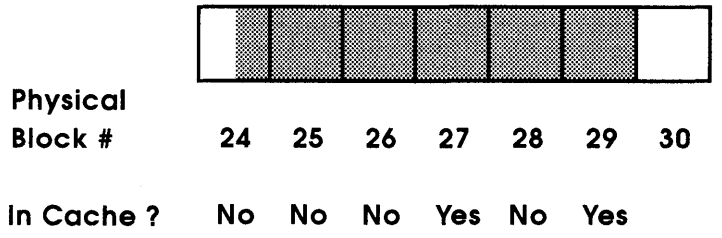

Figure 14. Blocking Factors and Cache-Buffering

5.7 Synchronization Modes

Because of the buffer cache, a volume might not always contain the most recent data. The data in a cache buffer miht have been modified, but not written to disk. If a hardware failure occurs before the disk is updated, the data will be lost.

A similar situation can arise with the system data structures used by pHILE+ to manage a volume (for example, FCBs, FATs, bit maps, and so forth). To reduce the number of disk accesses required during normal operation, copies of certain system data structures normally residing on volumes are maintained in memory. In this case, if a hardware failure occurs before pHILE updates a volume, then the volume will be corrupted.

To deal with these situations, and at the same time to accommodate different application requirements, pHILE+ provides three synchronization modes that dictate when a volume is updated. The synchronization mode is selected when a volume is mounted. The three possible modes are described in Table 1..

Table 1. Possible Modes for Synchronization

Mode	Effect
Immediate-Write	All changed data is flushed immediately
Control-Write	Flush only control data that changed
Delayed-Write	Flush data only as required

5.7.1 Immediate-Write Mode

When a volume is mounted with the immediate-write mode, cache buffers and in-memory system data structures are flushed (that is, written to the volume) whenever they are modified.

Immediate-write mode is equivalent to calling **sync_vol()** (explained below) after every pHILE+ operation. Unfortunately, system throughput can be significantly impacted since every write operation results in at least two I/O transactions: one for a cache buffer and one for system data. When using this mode, you should avoid writing less than a block of data with one **write_f()** system call. You should collect data in a local buffer and write at least one block at a time.

5.7.2 Control-Write Mode

When a volume has been mounted with control-write mode, every time an in-memory system data structure is modified, it is flushed to disk. For example, if the contents of a File Control Block is changed, it is flushed. User data, however, is not flushed immediately and may linger in a cache buffer for an indefinite period of time.

Control-write mode provides the same level of volume integrity as immediate-write mode, but provides less protection for your data in the event of a system failure. Its use, however, can significantly improve throughput. The difference is most dramatic when the application is performing **write_f()** operations involving small numbers of bytes.

5.7.3 Delayed-Write Mode

When a volume has been mounted with delayed-write mode, pHILE+ flushes memory-resident data only when required by normal operation. File Control Blocks are flushed only when a file is closed or a volume is synchronized. Cache buffers are flushed only when they are reused, a volume is synchronized, or a volume is unmounted.

The delayed-write mode is the most efficient of the three modes since its minimizes I/O. When using this mode, however, a system failure may leave a volume with inconsistent system data structures and old user data.

Delayed-write mode is a reasonable choice when high throughput is required. Normally, using the **sync_vol()** system call periodically is sufficient to maintain a consistent volume.

5.7.4 sync_vol

The **sync_vol()** system call copies the contents of the cache buffers and all in-memory system data structures to a volume. **sync_vol()** is automatically executed when a volume is unmounted.

It is not needed for a volume if the volume is mounted with immediate write mode.

5.8 pHILE+ Formatted Volumes

This section discusses how pHILE+ formatted volumes are organized and the special system calls available only for pHILE+ formatted volumes.

5.8.1 How pHILE+ Formatted Volumes Are Organized

As mentioned in Section 5.5, "Blocking/Deblocking," a pHILE+ formatted volume consists of a sequence of logical blocks. Several blocks per volume are dedicated to hold management information for the volume. These blocks are accessed directly by pHILE+ without going through normal file operations. The management blocks are defined as follows:

BOOTLOAD The first and second blocks (0 and 1) are never used by pHILE+. They are reserved in case a bootstrap loader is needed for the volume.

ROOTBLOCK Block 2 is always used as the root block for a volume. This block contains all information needed by pHILE+ to locate other vital information on the volume.

ROOTDIR Block 3 is always used to hold the first block of the root directory for the volume. As the root directory grows, additional blocks are allocated dynamically as required.

BITMAP This contiguous sequence of blocks is used to hold the bitmap for the volume, which uses bits to indicate what blocks are free. Its size and location are determined by parameters that you supply when you initialize the volume.

FLIST This contiguous sequence of blocks is used to hold the file descriptors for the volume. It is positioned immediately following the bitmap. Its size is determined by parameters you supply when you initialize a volume.

Thus, a volume has four initial data structures containing vital internal management data. Before a volume can be used, it must be initialized using the **init_vol()** call, described in the *pSOSystem Programmer's*

Reference manual. **init_vol()** builds the root block, the root directory, the bitmap, and the FLIST structures on the volume. See the *pSOSystem Programmer's Reference* manual for C language definitions of these data structures.

The bitmap can be placed anywhere on a volume and it is always followed by the FLIST. They need not be contiguous with the root block, root directory or any other data structure on the volume. Since the bitmap is used during write operations, and FLIST is used extensively during all file creation and connection, overall volume access can be improved by careful placement of these structures.

5.8.1.1 The Root Block

The root block is the starting point from which pHILE+ locates all other data on the volume. For this purpose, it contains the:

BITMAP_ADDRESS	The starting block number of the volume bitmap
FLIST_ADDRESS	The starting block number of FLIST
DATA_ADDRESS	The starting block number of data space (See section 5.8.1.5.)

In addition, the root block contains the following information about the volume:

INIT_TIME	The time and date of volume initialization
VOLUME_NAME	The volume label
VOLUME_SIZE	The volume size in blocks
NUMBEROF_FD	The number of file descriptors (that is, the FLIST size)
VALIDATE_KEY	Confirmation of successful volume initialization

5.8.1.2 The Root Directory

The volume's root directory is a directory file that forms the starting point from which pHILE+ locates all other files on a volume. From the root directory emanates the tree structure of (sub)directories and ordinary files. In the simplest case, the root directory may contain only ordinary

files, thus yielding a one-level directory structure common in less sophisticated file systems.

Immediately after a volume has been initialized, its root directory contains two files: **FLIST.SYS**, which is the volume's list of file descriptors, and **BITMAP.SYS**, which is the volume's map of occupied blocks.

As with any user file, ordinary or directory, the root directory is expanded automatically by pHILE+, as required. For directory files, such expansion occurs one block at a time, and the blocks are generally not contiguous. Contiguous expansion of directory files can be achieved using the **annex_f()** function described in the *pSOSystem Programmer's Reference* manual.

5.8.1.3 The Volume Bitmap

A volume's bitmap is actually a system file. It is read-only; it performs the critical function of tracking the usage of each block on the volume. One bit is used to tag each block in the volume. If a block is allocated to a file, then the corresponding bit is set to 1. If a block is free, the corresponding bit is 0.

The size of the bitmap is determined by the size of the volume. Thus, for example, if the volume has 32K blocks, then the bitmap uses 32K bits or 4 Kbytes. If block size is 1 Kbyte, then 4 blocks are allocated for this bitmap.

Immediately after a volume has been initialized, its bitmap shows blocks used by the bootloader, the root block, the bitmap itself, and **FLIST.SYS**.

The bitmap can be read as *<volume>*/**BITMAP.SYS**. This file is write-protected, and hence cannot be written to directly or deleted.

5.8.1.4 The File Descriptor List

Every file, whether it is an ordinary or directory file, requires a control structure called a file descriptor (FD). Each volume contains its own list of file descriptors, called the FLIST, which is stored in a contiguous sequence of blocks. More details about file descriptors are in section 5.8.2.2.

You specify the number of file descriptors in the FLIST when you initialize a volume. Each file descriptor is 128 bytes long. Therefore, if

the number of file descriptors specified is 100, the FLIST occupies 12800 bytes, or 13 blocks if the block size is 1 Kbyte.

Note that if the number of file descriptors on a volume is specified as *n*, then the maximum number of user-created files that can exist on the volume is *n*. The number of file descriptors created will actually be (*n* + 4), since four internal system files are always present: the root directory (**/**), **/BITMAP.SYS**, **/FLIST.SYS**, and a reserved null file. These system files are write-protected, and cannot be written to directly or deleted.

5.8.1.5 Control and Data Block Regions

pHILE+ formatted volumes recognize two types of file blocks: control blocks and data blocks. Control blocks contain pHILE+ data structures such as:

- The bootload (blocks 0 and 1)
- The root block (block 2)
- The bitmap
- The FLIST
- All directory file blocks
- Indirect and index blocks

Indirect and index blocks are used with extent maps and are explained in section 5.8.2.7.

Data and control blocks can be either intermixed or partitioned. Partitioning control and data blocks is a unique feature of pHILE+ formatted volumes and makes pHILE+ capable of working with write-once devices. When a partition is used, the logical address space of a volume is divided into two regions: one for control blocks and one for data blocks. Using this method, control blocks can be temporarily maintained on an erasable media while data blocks are written on a write-once device. After the data partition of a volume is filled, the information from the control blocks that had been on erasable media can be transferred to the write-once device, where it is permanently recorded.

Intermixing control and data blocks means that your data and pHILE+ data structures will be written randomly on a device.

The manner in which control and data blocks are organized on a volume is determined when the volume is initialized. One of the input parameters to **init_vol()** specifies the starting block number of the volume's data blocks. If zero is specified, then the data and control blocks are intermixed. Otherwise, data blocks begin at the specified block. For example, if a data block starting number of 200 is specified on a volume containing 5000 blocks, then blocks 2 - 199 (recall blocks 0 and 1 are not used by pHILE+) are control blocks and blocks 200 - 4999 are data blocks.

5.8.2 How Files Are Organized

A file is a collection of blocks that contain data, a file descriptor that contains control information, and an entry in a parent directory file.

The following sections outline how files are constructed and how data in them is used.

5.8.2.1 The File Number

Externally, a file is specified by its pathname. Internally, pHILE+ converts this pathname into a corresponding file number, which is indexed. With this file number, pHILE+ accesses a file descriptor, and uses its content to perform the necessary operations on the file. You cannot use the file number externally as a file ID. A call such as **create_f()**, for example, returns an external file ID, not the internal, proprietary file number.

5.8.2.2 The File Descriptor

Each file descriptor is 128 bytes and contains the following information:

- The logical file size in bytes
- The physical file size in blocks
- The file type: directory or ordinary, system or data
- The time of last modification
- The file's expansion unit
- The file's extent map.

5.8.2.3 File Types

There are two type attributes associated with a file. A file may be an ordinary or a directory file, and it may be a system file or a data file. Ordinary and directory files were discussed above.

System files are created by pHILE+ when a volume is initialized. There are three system files per volume:

/BITMAP.SYS	The volume's bitmap
/FLIST.SYS	The volume's FLIST
/	The volume's root directory

Since system files contain vital data structures, they are protected against user removal and modification. Reading, however, is allowed.

5.8.2.4 Time of Last Modification

pHILE+ maintains the time at which a file was last modified. This field is initialized when a file is created; thereafter it is updated whenever a file is written, or when blocks are annexed to the file.

5.8.2.5 The File Expansion Unit

If a **write_f()** operation extends past the current physical size of a file, pHILE+ will automatically expand the file to hold the new data. This type of file expansion is governed by the following considerations.

When a file is created, you supply a parameter called an *expansion unit* that determines the minimum expansion increment to use during **write_f()** operations. This parameter specifies the minimum number of physically contiguous blocks pHILE+ attempts to allocate when additional space is required by file. This is a lower-bound number, since the number of blocks allocated is actually determined by either the expansion unit, or the number of blocks needed to satisfy the current **write_f()** operation, whichever is greater.

5.8.2.6 Extents

A file is treated simply as a sequence of logical blocks. Each such block corresponds to a physical block on the volume. Since the physical blocks that comprise a file may be scattered throughout a volume, pHILE+

implements a structure called an *extent* to keep track of a file's blocks, and hence its data.

An extent is a sequence of physically contiguous blocks. An extent consists of one or more blocks; similarly, a file with data consists of one or more extents.

A file can acquire an extent in one of two ways:

- During a **write_f()** operation, when a file is expanded; or
- During an **annex_f()** operation

These operations also might *not* produce a new extent, since pHILE+ may merge the newly allocated blocks into an existing extent (logically the last extent) if the new blocks are contiguous with that extent.

An extent is described by an extent descriptor:

< starting block number, number of blocks >

which identifies the physical address of the blocks that make up the extent.

5.8.2.7 The Extent Map

The *extent map* for a file is a list of its extent descriptors. For reasons of efficiency, this map is organized by layers of indirection.

The first 10 extent descriptors are located in the file's file descriptor. Additional extent descriptors, when needed, are stored in indirect blocks. Each indirect block is a physical block that contains up to *n* extent descriptors. Since an extent descriptor is 8 bytes, the number *n* of extent descriptors that can be held in an indirect block is (blocksize / 8). For example, if blocksize is 1 Kbyte, then *n* is 128. Indirect blocks are allocated as needed for each file.

Each indirect block is addressed via an indirect block descriptor which is also a pair of words:

< starting block number, last logical block number + 1 >

where the first item is a physical block number, and the second item is the logical number (+ 1) of the last block contained in this indirect block of extent descriptors. This last number is useful for quickly determining whether an indirect block needs to be searched while locating a particular logical block within a file.

The indirect block descriptor for the first indirect block, if needed, is held in a file descriptor. If more than one indirect block is needed, as in the case of rather large and scattered files, then the second through (*n* + 1)th indirect block descriptors are held in an index block.

If allocated, this index block will contain up to *n* indirect block descriptors. Again, since each indirect block descriptor is 8 bytes long, the number *n* of indirect block descriptors in the index block is equal to (blocksize / 8). For example, if blocksize is 1 Kbyte, then this number will be 128. The physical block address of the index block is contained in a file descriptor. A file can have only one index block.

The structure of the extent map ensures that, in the worst case, no more than two block accesses are needed to locate an extent descriptor. Moreover, the cache buffers will tend to retain frequently used index and indirect blocks.

This extent map structure clearly favors file contiguity. For example, if a file can be covered in fewer than 10 extents, then access to any of its data can be accomplished via the file descriptor alone.

The extent map will hold up to [*n* * (*n* + 1) + 10] extents, where *n* is (blocksize / 8), as above. For example, if blocksize is 1 Kbyte, then the maximum number of extents per file is [(128 * 129) + 10], or 16522. In the worst case of 1 block per extent, a file can contain 16522 blocks, or 16 megabytes of data. However, because pHILE+ contains both implicit and explicit features to "cluster" many blocks into a single extent, the number of extents required to map a file is usually very much smaller. In fact, even for a very large file, the number of extents needed to map the file rarely exceeds 100.

Figure 15 illustrates an example of an extent map layout.

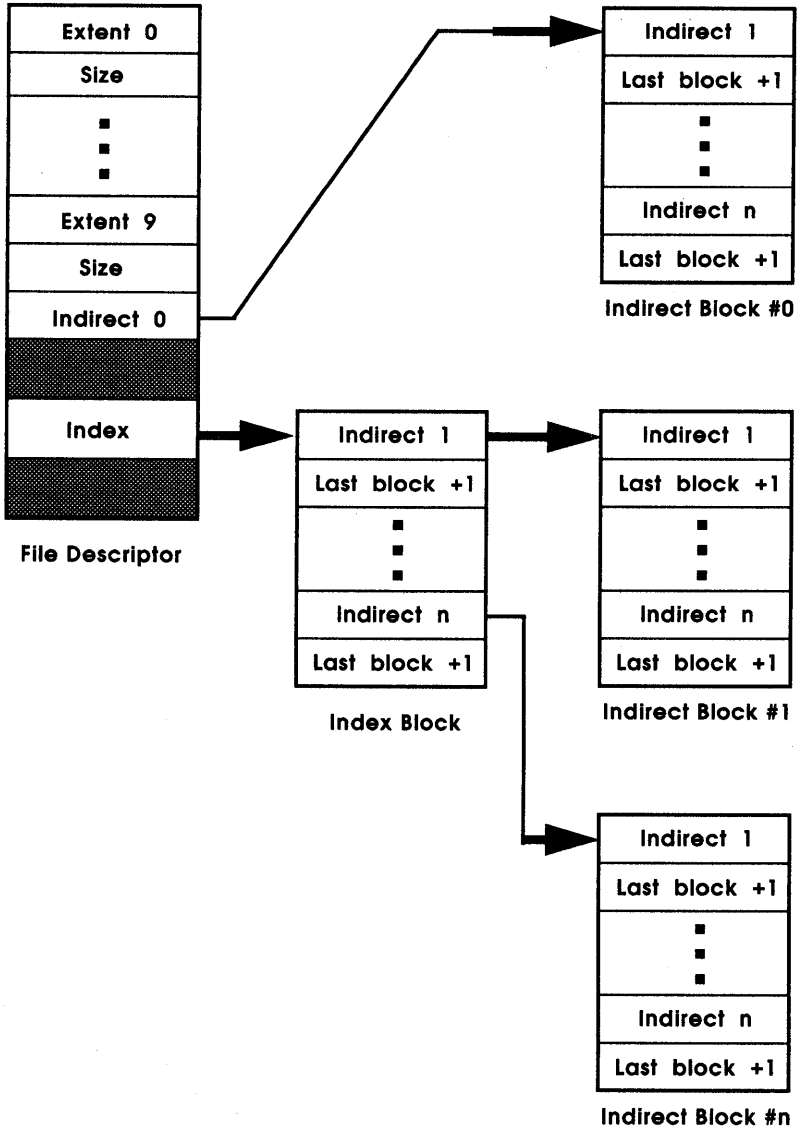

Figure 15. The Layout of an Extent Map

5.8.3 Data Address Mapping

pHILE+ allows you to access file content down to individual bytes. For each file access, pHILE+ performs a number of address translations that convert or map your stretch of data into a volume block or blocks.

As an example of file content access, consider a file with three extents. Assume its file descriptor's extent map looks like the following:

(060,5)
(789,2)
(556,1)

That is, the file has 8 blocks. Assume that block size is 1 Kbyte. If a read call requests 100 bytes, starting at byte number 7000, the request is processed by pHILE+ as follows:

1. Byte 7000 divided by 1024 = 6, remainder = 856.
2. Logical file block 6 is needed, since blocks are numbered from 0.
3. According to extent map, block #6 is the 2nd block in the extent (789,2).
4. pHILE+ calls your driver to read volume block #790.
5. pHILE+ extracts bytes 855 to 955 from the 1024 bytes that were read in.

5.8.4 Block Allocation Methods

Since blocks are the basic unit of the pHILE+ volume, block allocation algorithms are extremely important to system throughput. Blocks must be allocated whenever:

- A **write_f()** extends the logical size of a file beyond the file's physical size.
- An **annex_f()** call is made.
- A new block must be added to a directory to accommodate a new entry. This can happen on a **create_f()**, **make_dir()**, or **move_f()** call.
- An indirect or index block must be added when a new extent is added to a file. This can happen whenever blocks are allocated -- for whatever reason.

When more blocks are needed, pHILE+ first determines the allocation size. This is the ideal size of the extent to be allocated. The allocation size for each case above is determined as follows:

Case 1: write_f() Extends a File

When extending an ordinary file to write data from a **write_f()** call, the allocation size is the larger of the number of blocks needed for the data and the expansion unit that you specified when the file was created. For example, assume that a **write_f()** call requires two blocks. If the file was created with an expansion unit of five blocks, then the allocation size will be five blocks. On the other hand, if the file's expansion unit is one, then the allocation size will be two blocks.

Case 2: annex_f() Extends a File

The allocation size is a parameter of the **annex_f()** call and is thus provided by the calling task.

Case 3: A New Entry Extends A Directory File

Directories have the following properties:

- They grow one entry at a time;
- Each entry is 16 bytes long; and,
- There is no expansion unit associated with a directory

For all of these reasons, the directory allocation size is always one block.

Case 4: An Indirect or Index Block Is Needed

These are always single blocks, so the allocation size is one block. Indirect and index blocks are explained below.

After selecting the allocation size, pHILE+ chooses the block type. Ordinary files use data blocks, while pHILE+ data structures use control blocks.

The block type is used to decide where in the volume to search for free space. If the volume was partitioned into data and control regions during initialization, which is explained in more detail below, only the relevant portion of the volume will be used.

The search does not always start with the first block in the appropriate region. Rather, pHILE+ will start searching in the bitmap of the block last referenced. This increases the chance of scanning a block in the cache, and thus enhances throughput.

The search involves locating the first unused extent containing at least the required number of blocks. This search can have three outcomes:

1. A sufficiently large extent is found and allocated, in which case the search is successfully completed. If the length of the extent is greater than the allocation size, the extent will be split.
2. No extents equal to or greater than the allocation size are found. In this case, pHILE+ will allocate the largest remaining extent in the appropriate region. If the calling function is **annex_f()**, the number of blocks actually allocated is returned to the caller. If a **write_f()** is executed, a new allocation size is calculated (depending on the number of blocks not yet allocated) and the operation is repeated. That way, one **write_f()** call can add several extents to a file.
3. The volume is full; that is, there are no free blocks. In this case, a "volume full" error is returned to the calling task.

The time to read and write to a file depends on how fragmented the file is. A file fragmented into many small and scattered extents will take more time to access than a file consisting of fewer and larger extents. If a file can be compacted into 10 or fewer extents, then all of the file's data blocks can be identified using an extent map stored in the File Control Block. This is the optimal case. If a file has more than 10 extents, indirect blocks or index blocks must be used, which reduces access times.

Some attention should be given to a file's expansion unit specification, which is described in section 5.8.2.5. A larger expansion unit results in higher throughput, but may waste disk space, since some blocks may not be used. On the other hand, a smaller expansion unit uses disk space more efficiently, but may cause fragmentation. This fragmentation will be a function of:

- The average number of bytes written per **write_f()**;
- The number of **annex_f()** calls used; and,
- Concurrent file activity; that is, how many tasks are using the volume at the same time.

When pHILE+ needs to add blocks to a file, it always checks to see if the new blocks can be merged into the last extent used.

5.8.5 How Directories Are Organized

Directories implement the hierarchical file structure of pHILE+. A volume's directory tree structure is built on top of, but also out of, the basic data file structure. That is, directory files are treated in almost all respects as ordinary data files. Directory files hold data about their children, and the parent of a directory will hold data about the directory. A directory file contains an array of entries. Each entry describes a file in the directory. An entry is nothing more than a 2-tuple, as follows:

Entry: < *filenumber*, *filename* >.

filenumber is the number of the file and *filename* is its name. Each directory entry uses 16 bytes, so if the block size is 1 Kbyte, one block can store 64 entries.

When a file is created, pHILE+ assigns it a file descriptor in the volume's FLIST, described below, and makes an entry in the directory file to which it belongs.

5.8.6 Logical and Physical File Sizes

Files occupy an integral number of storage blocks on the device. However, pHILE+ keeps track of the length of a file in bytes. Unless the length of a file is an exact multiple of the block size, the last block of the file will be partially used. There are therefore two sizes associated with every file: a logical size and a physical size.

The logical size of a file is the number of data bytes within the file that you can access. This size automatically increases whenever data is appended to the file, but never decreases.

The physical size of a file corresponds to the number of blocks currently allocated to the file. Thus the logical and physical sizes of a file are generally different, unless a file's logical size happens to exactly fill the number of physical blocks allocated to the file. As with its logical size, a file's physical size never decreases, except, of course, when it is deleted.

5.8.7 Special Services

This section discusses those services available after you create a pHILE+ formatted volume. These services are not available with MS-DOS or NFS volumes.

5.8.7.1 get_fn, open_fn

Each time a file is opened, the pathname must be parsed and the directories searched. If the pathname traverses many levels of the directory tree, or if any directory in the path contains a large numbers of files, then a directory search can be time-consuming. Most applications open files infrequently, and the directory search time in such cases is unimportant. However if the same file must be frequently opened and closed, the parsing and searching overhead can be substantial.

On pHILE+ formatted volumes, an alternate method of opening a file, **open_fn()**, bypasses all parsing and directory searching. Rather than providing a pathname, the calling task can provide instead the file number. The **get_fn()** call is used to obtain the file number. **get_fn()** accepts a pathname as input and returns the file number of the corresponding file. **get_fn()** followed by an **open_fn()** is functionally equivalent to an **open_f()** call. If the file is to be opened many times, it is more efficient to call **get_fn()** once, and then use **open_fn()** whenever the file must be opened.

A second and less obvious advantage of **get_fn()** and **open_fn()** involves reusing pathnames. Often a pathname must be saved so a file can be reopened later. If a file is deeply nested, its pathname can be quite long and may consequently require a significant amount of memory for storage. Even worse, if a saved pathname is expressed relative to a current directory and the current directory changes before the file is reopened, the operation will fail or the wrong file will be opened.

In these cases, the pathname can instead be converted into a file number. The file can be (re)opened at a later time, independently of the current directory.

5.8.7.2 annex_f

write_f() operations will automatically add new blocks to a file as required, but the blocks added often are not contiguous. This situation can be controlled to some extent on pHILE+ formatted volumes by using a larger file expansion unit. For even more efficient, contiguous grouping, the **annex_f()** function may be used to manually allocate or expand a file's physical size, in anticipation of new data.

Call **annex_f()** by passing the number of contiguous blocks you wish to add to a file, known by a file ID; the call will return the number of blocks added. **annex_f()** does nothing, however, to the logical size of the file --

see the cautions in the description of the call. If a file's final size can be estimated in advance, then **annex_f()** may be used to allocate a single contiguous extent for the file immediately after its creation. So long as subsequent write operations do not extend past this size, the file will be truly contiguous. If the file must be expanded, then this may be left implicitly to **write_f()**, or performed explicitly using additional **annex_f()** operations.

5.8.7.3 lock_f

pHILE+ allows a single file to be opened and accessed by more than one task simultaneously. Concurrent read access is generally quite safe; however, if one or more tasks perform write operations (concurrent update), then it may be necessary for such tasks to secure exclusive access to all or part of the file.

The **lock_f()** function allows a task to lock a specified region of a file. As long as the lock is in effect, pHILE+ will prevent all other file connections from reading, writing or locking that region of the file, thus providing exclusive access to a single connection.

lock_f() requires two parameters. The first is the position of the first byte to lock. The second is the number of bytes to lock. A lock may start and/or end beyond both the physical or logical end of a file. This allows a lock to anticipate future expansion of a file. Thus, **lock_f()** can be used to prevent all other connections to the file from:

- Modifying or appending any data in the locked region of the file, and
- Reading any data in, or being appended to, the locked region of the file.

When a lock is in place, the locked region can be accessed only by the task that placed the lock and then only via the file ID with which the lock was placed.

Each connection to a file may lock only one region of a file at any time. If a task needs to lock two different parts of a file simultaneously, then it must open the file twice to obtain a second connection (via a different file ID).

If a **lock_f()** call is issued through a connection that has an existing lock, then the existing lock is automatically removed and replaced by the new lock. This lock replacement takes place as an atomic operation. That is, the existing lock is removed, and the new lock is set in a single operation.

This precludes, in the case that the old and new regions overlap, any opportunity for another task to access -- or even worse, lock -- the overlapped region during the replacement window.

To remove an existing lock, simply replace it with a new lock of length zero, using the same file ID.

A lock prevents offending **read_f()**, **write_f()**, and **lock_f()** operations only. It does not prevent another task from adding blocks to a file with the **annex_f()** call. Nor does it prevent access to the file's data via the **read_vol()** and **write_vol()** calls.

5.8.7.4 Direct Volume I/O

While a volume's data is usually accessed through the directory organization provided by pHILE+, certain applications may need to access data via its logical address on the volume.

Two pHILE+ system calls, **read_vol()** and **write_vol()**, allow you to access data on a pHILE+ formatted volume by block address. Any number of bytes may be accessed, beginning at any byte within any logical block on a volume.

These calls provide two advantages compared to calling the appropriate device driver directly, which bypasses pHILE+ entirely. First, if the volume has been mounted with some synchronization mode other than immediate write, data recently written to the volume may still be memory-resident, not having yet been flushed to the device. Calling the driver directly would not read the latest copy of such data. Worse, data written directly to the volume could be overwritten by cache data and thus lost entirely.

read_vol() and **write_vol()** can read/write portions of a block. All the necessary caching and blocking/deblocking will be performed by pHILE+ as required. Thus **read_vol()** and **write_vol()** allow a device to be accessed as a continuous sequence of bytes without regard for block boundaries.

5.8.8 Restarting and Deleting Tasks That Use pHILE+

During normal operation, pHILE+ internally allocates and holds resources on behalf of calling tasks. Some resources are held only during execution of a service call, while others are held indefinitely based the state of the task (for example when files are open). The pSOS+ service

calls **t_restart()** and **t_delete()** asynchronously alter the execution path of a task and present special problems relative to management of these resources.

This section discusses deletion- and restart-related issues in detail and presents recommended ways to perform these operations.

5.8.8.1 Restarting Tasks That Use pHILE+

pSOS+ allows a task to be restarted regardless of its current state. The restart operation has no effect on currently opened files. All files remain open and their L_ptr's are unchanged.

It is possible to restart a task while the task is executing code with the pHILE+ component. Consider the following example:

1. Task A makes a pHILE+ call.
2. While executing pHILE+ code, task A is preempted by task B.
3. Task B then restarts task A.

In such situations, pHILE+ correctly returns resources as required. However, a file system volume may be left in an inconsistent state. For example, if **t_restart()** interrupts a **create_f()** operation, a file descriptor (FD) may have been allocated but not the directly entry. As a result, an FD may be permanently lost. **t_restart()** detects potential corruption and returns the warning code 0x0D. When this warning code is received, **verify_vol()** should be used to detect and correct any resulting volume inconsistencies.

5.8.8.2 Deleting Tasks That Use pHILE+

To avoid permanent loss of pHILE+ resources, pSOS+ does not allow deletion of a task that is holding any pHILE+ resource. Instead, **t_delete()** returns error code 0x18, which indicates that the task to be deleted holds pHILE+ resources.

The exact conditions under which pHILE+ holds resources are complex. In general, any task that has made a pHILE+ service call may hold pHILE+ resources. **close_f(0)**, which returns all pHILE+ resources held by the calling task, should be called by the task to be deleted prior to calling **t_delete()**.

pNA+ and pREPC+ also hold resources which must be returned before a task can be deleted. These resources are returned by calling **close(0)**

and **fclose(0)** respectively. Since pREPC+ calls pHILE+, and pHILE+ calls pNA+ (if NFS is in use), these services must be called in the correct order. Below is a sample code fragment that a task can use to delete itself:

```
fclose(0);        /* return pREPC+ resources */
close_f(0);       /* return pHILE+ resources */
close(0);         /* return pNA+ resources */
t_delete(0);      /* and commit suicide */
```

Obviously, close calls to components not in use should be omitted.

Since only the task to be deleted can make the necessary close calls, the simplest way to delete a task is to restart the task and pass arguments requesting self deletion. Of course, the task being deleted must contain code to handle this condition.

(Blank Page)

6 pREPC+ ANSI C Library

6.1 Introduction

Most C compilers are delivered with some sort of run-time library. These run-time libraries contain a collection of pre-defined functions that can be called from your application program. They are linked with the code you develop when you build your application. However, when you attempt to use these libraries in a real-time embedded system, they encounter one or more of the following problems:

- The library functions are not reentrant and therefore do not work in a multitasking environment.
- It is the user's responsibility to integrate library I/O functions into the target environment, a time-consuming task.
- The library functions are not compatible with a published standard, resulting in application code that is not portable.

pREPC+ solves all of the problems mentioned above. First, it is designed to work with the pSOS+ Real-Time Multitasking Kernel and the pHILE+ File System Manager, so all operating system dependent issues have been addressed and resolved. Second, it is designed to operate in a multitasking environment, and finally, it complies with the C Standard Library specified by the American National Standards Institute.

6.2 Functions Summary

pREPC+ provides more than 85 run-time functions. Following the conventions used in the ANSI X3J11 standard, these functions can be separated into 4 categories:

- Character Handling Functions
- String Handling Functions
- General Utilities
- Input/Output Functions

The Character Handling Functions provide facilities for testing characters (for example, is a character a digit?) and mapping characters (for example,. convert an ASCII character from lowercase to uppercase).

The String Handling Functions perform operations on strings. With these functions you can copy one string to another string, append one string to another string, compare two strings, and search a string for a substring.

The General Utilities provide a variety of miscellaneous functions including allocating and deallocating memory, converting strings to numbers, searching and sorting arrays, and generating random numbers.

I/O is the largest and most complex area of support. The I/O Functions include character, direct, and formatted I/O functions. I/O is discussed in Section 6.3, "I/O Overview."

Detailed descriptions of each function are provided in the *pSOSystem Programmer's Reference* manual.

6.3 I/O Overview

There are several different levels of I/O supported by the pREPC+/ pSOS+/pHILE+ environment, providing different amounts of buffering, formatting, and so forth. This results in a layered approach to I/O, since the higher levels call the lower levels. The main levels are shown in Figure 16.

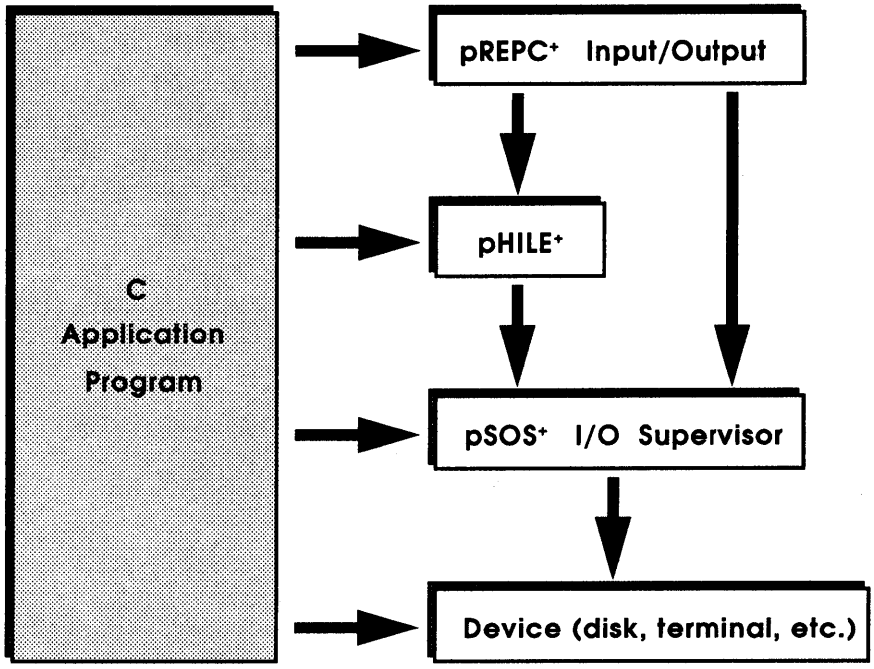

Figure 16. I/O Structure of pREPC+

The pREPC+ I/O functions provide a uniform method for handling all types of I/O. They mask the underlying layers and allow application programs to be hardware and device independent. A user application can, however, call any of the layers directly, depending on its requirements.

The lowest, most primitive way of doing I/O is by directly accessing the hardware device involved, for example a serial channel or a disk controller. Programming at this level involves detailed knowledge of the device's registers, etc. Although all I/O eventually reaches this level, it is almost never part of the application program, as it is too machine-dependent.

The next step up from the actual device is to call a device driver. Under pSOS+, all device drivers are called in a similar fashion, via the pSOS+

I/O Supervisor, which is explained in Chapter 7, "I/O System." For reading and writing, all that is generally required is a pointer to the buffer to read into or write from, a character count, and a way to identify the device being used.

The pSOS+ I/O Supervisor provides the fastest, most direct route for getting a piece of data to a device. In some cases, this is the best way. Generally, however, it is better to use the pREPC+ direct, character, or formatted I/O services.

pHILE+ manages and organizes data as sets of files on storage devices and in turn does all of the actual I/O. The pHILE+ I/O path depends on the type of volume mounted and is described in detail in Chapter 5, "pHILE+ File System Manager."

pHILE+ services (such as **open_f** and **write_f**) can be called directly. However, if you use the pREPC+ file I/O functions, which in turn call pHILE+, your application code will be more portable.

The pREPC+ direct I/O and character I/O functions read and write sequences of characters. The formatted I/O functions perform transformations on the input and output and include the familiar **printf()** and **scanf()** functions.

6.3.1 Files, Disk Files, and I/O Devices

Under pREPC+, all I/O is directed to and from "files." pREPC+ divides files into two categories: I/O devices and disk files. They are treated as similarly as possible, but there are intrinsic differences between the two.

Disk files are part of a true file system managed by pHILE+. There is a file position indicator associated with each disk file, which marks the current location within the file. It is advanced whenever data is read from or written to the file. In addition, it can be changed via function calls.

pHILE+ manages three types of volumes. These are pHILE+ formatted volumes, MS-DOS volumes, and NFS (Network File System) volumes. pREPC+ does not distinguish between the underlying volume types and therefore works equally well with all three volume types. However, there are a number of small differences between the various volumes that may affect the results of certain pREPC+ functions. Function descriptions

indicate those cases where the volume type may affect function results and how those functions would be affected.

I/O devices correspond to pSOS+ logical devices, and are usually associated with devices such as terminals or printers. From an application's standpoint, their main difference from disk files is that they have no position indicator. Data being read from or written to an I/O device can be thought of as a continuous stream.

When reading and writing disk files, pREPC+ calls pHILE+, which in turn calls the pSOS+ I/O Supervisor. When reading and writing I/O devices, pREPC+ calls the pSOS+ I/O Supervisor directly.

Before a file (a disk file or an I/O device) can be read or written, it must be opened using **fopen()**. One of the **fopen()** function's input parameters is a name that specifies the file to open. Disk files are designated by pHILE+ pathnames, while I/O devices are identified by pSOS+ logical device numbers.

Examples:

3.2 designates an I/O device with logical device number 3.2.

3.2/abcd designates a disk file stored on logical device 3.2.

abcd designates a disk file in the current directory.

When **fopen()** opens a disk file, it generates a pHILE+ **open_f()** system call. When it opens an I/O device, **fopen()** calls the pSOS+ **de_open()** service. Regardless of whether **fopen()** opens an I/O device or a disk file, it allocates a **FILE** data structure, which is discussed in section 6.3.2.

6.3.2 File Data Structure

As mentioned in the previous section, when a file is opened, it is allocated a data structure of type **FILE**. In pREPC+ this is a 32-bit address of a pREPC+ file structure. **fopen()** returns a pointer to this allocated data structure. All file operations require the pointer to this structure as an input parameter to identify the file. If it is not explicitly given, it is implied, as in the case of functions which always use the standard output device (See section 6.3.4).

The **FILE** data structure is used to store control information for the open file. Some of the more important members of this structure include the address of the file's buffer, the current position in the file, an end-of file

(EOF) flag, and an error flag. In addition, there is a flag that indicates whether the file is a disk file or an I/O device.

Some of these fields have no meaning for I/O devices, such as the position indicator.

6.3.3 Buffers

Open files normally have an associated "buffer" that is used to buffer the flow of data between the user application and the device. By caching data in the buffer, pREPC+ avoids excessive I/O activity when the application is reading or writing small data units.

When first opened, a file has no buffer. Normally a buffer is automatically assigned to the file the first time it is read or written. The buffer size is defined by the entry **LC_BUFSIZ** in the pREPC+ Configuration Table. pREPC+ allocates the buffer from pSOS+ region 0. If memory is not available, the calling task may block based on the values in the pREPC+ configuration table entries **LC_WAITOPT** and **LC_TIMEOPT**. If a buffer cannot be obtained an error is returned to the read or write operation.

Note that if the default buffer assigned by pREPC+ is not appropriate for a particular file, a buffer may be supplied directly by calling the **setbuf()** or **setvbuf()** functions.

A special case arises when a file is assigned a buffer of length 0. This occurs if **LC_BUFSIZ** is zero, and as an option to the **setvbuf()** call. In this case, no buffer is assigned to the file and all I/O is unbuffered. That is, every read or write operation through pREPC+will result in a call to a device driver of pHILE+ as the case may be.

Finally, note that the three standard files, **stdin**, **stdout**, and **stderr**, are not affected by the value of **LC_BUFSIZ**. See section 6.3.5 for a discussion of the default buffering of these three files.

6.3.4 Buffering Techniques

This section describes the buffering techniques used by pREPC+. There are two cases to consider, writing and reading.

On output, data is sent to the file's buffer and subsequently transferred (or "flushed") to the I/O device or disk file by calling a pSOS+ device driver (for an I/O device) or pHILE+ (for a disk file). The time at which a

buffer is flushed depends on whether the file is *line-buffered* or *fully-buffered.* If line-buffered, the buffer is flushed when either the buffer is full or a newline character is detected. If fully-buffered, the buffer is flushed only when it is full. In addition, data can be manually flushed, or forced, from a buffer at any time by calling the **fflush()** function. Finally, the buffer is flushed if the data direction is reversed; that is, a read operation is performed immediately after a write.

By default, I/O devices are line-buffered, whereas disk files are fully-buffered. This can be changed after a file is opened by using the **setbuf()** or **setvbuf()** functions.

When reading, pREPC+ retrieves data from a file's buffer. When attempting to read from an empty buffer, pREPC+ calls either a pSOS+ driver or pHILE+ to replenish its contents. When attempting to replenish its internal buffer, pREPC+ reads sufficient characters to fill the buffer. The pSOS+ driver or pHILE+ may return fewer characters than requested. This is not necessarily considered as an error condition. If zero characters are returned, pREPC+ treats this as an EOF condition.

Note that the buffering provided by pREPC+ adds a layer of buffering on top of the buffering implemented by pHILE+.

6.3.5 stdin, stdout, stderr

Three files are opened automatically for every task that calls pREPC+. They are referred to as the standard input device (**stdin**), the standard output device (**stdout**) and the standard error device (**stderr**). They can be disk files or I/O devices and are defined by entries in the pREPC+ Configuration Table. **stdin**, **stdout** and **stderr** are implicitly referenced by certain input/output functions. For example, **printf()** always writes to **stdout**, and **scanf()** always reads from **stdin**.

stdout and **stderr** are opened in mode **w**, while **stdin** is opened in mode **r**. Modes are discussed in the **fopen()** description given in the *pSOSystem Programmer's Reference* manual. Each file is assigned a 256 byte buffer. **LC_BUFSIZ** has no effect on the buffer size of these three files.

The buffering characteristics for **stdin** and **stdout** depend on the type of files specified for these devices. In the case of an I/O device, they are line-buffered. In the case of a disk file, they are fully-buffered. **stderr** is

an exception. Regardless of whether **stderr** is attached to a disk file or an I/O device, it is fully-buffered.

Like any other file, the buffer size and buffering technique of these files can be modified with the **setbuf()** and **setvbuf()** function calls.

pREPC+ attempts to open **stdin**, **stdout** and **stderr** for a task the first time the task issues a pREPC+ system call. If any of these files cannot be opened, pREPC+ calls the **k_fatal** service with a 0x3F03 error code as an input parameter.

When opened, the pathname of the files is obtained from the pREPC+ configuration table. Even though each task maintains a separate file structure for each of the three standard files, they all use the same **stdin**, **stdout**, and **stderr** device or file. This may not be desirable in your application. The **freopen()** function can be used to dynamically change the pathnames of any file, including **stdin**, **stdout**, and **stderr**, in your system. For example, to change the **stdout** from its default value of I/O device **00.00** to a disk file (**01.00/std_out.dat**) you would use the following function:

```
freopen("01.00/std_out.dat", "w", stdout);
```

When using **freopen** with the three standard files, two rules should be observed. First, the mode of the standard files should not be altered from their default values, and second, you should not use pathnames that include the strings "stdin", "stdout", or "stderr".

6.3.6 Streams

Streams is a notion introduced by the X3J11 Committee. Using the X3J11 Committee's terminology, a stream is a source or destination of data that is associated with a file. The Standard defines two types of streams: *text streams* and *binary streams*. In pREPC+, these are identical. In fact, in pREPC+, a stream is identical to a file. Therefore, we have chosen to continue using the more familiar term "file", rather than the term "stream" in this manual.

6.4 Memory Allocation

The following pREPC+ functions allocate blocks of memory:

calloc()
malloc()
realloc()

When any of these functions are called, pREPC+, in turn, calls the pSOS+ region manager by generating a **rn_getseg** call. pREPC+ always requests segments from Region 0. Therefore, you must reserve enough space in Region 0 for the memory required by your application and for the memory used by pREPC+ for file buffers (see section 6.3.3).

The **rn_getseg** call's input parameters include wait/nowait and timeout options. The wait/nowait and timeout options used by pREPC+ when calling **rn_getseg** are specified in the pREPC+ Configuration Table. Note that if the wait option is selected, it is possible for any of the functions listed above to result in blocking the caller. Also note that the number of bytes actually allocated by each **rn_getseg** call depends on Region 0's **unit_size**.

The following functions result in memory deallocation:

free()
realloc()
fclose()
setbuf()
setvbuf()

The **free()** function is called by a user for returning memory no longer needed. The remaining functions implicitly cause memory to be released. pREPC+ deallocates memory by generating a **rn_retseg** call to pSOS+.

Chapter 2, "pSOS+ Real-Time Kernel," contains a complete discussion of the pSOS+ region memory manager.

6.5 Error Handling

Most pREPC+ functions can generate error conditions. In most such cases, pREPC+ stores an error code into an internal variable called **errno** and return an "error indicator" to the calling task. Usually this error indicator takes the form of a negative return value. The error indicator for each function, if any, is documented in the individual function calls. Error codes are documented in the *pSOSystem Programmer's Reference* manual.

pREPC+ maintains a separate copy of **errno** for each task. Thus, an error occurring in one task will have no effect on the **errno** of another task. A task's **errno** value is initially zero. When an error indication is returned from a pREPC+ call, the calling task can obtain the **errno** value by referencing the macro **errno**. This macro is defined in the include file **<stdio.h>**. Note that the value of **errno** associated with a particular pREPC+ call is only valid as long as no other pREPC+ services are called by the task. This is true whether or not the second call generates an error.

pREPC+ also maintains two error flags for each opened file. They are called the *end-of-file flag* and the *error flag*. These flags are set and cleared by a number of the I/O functions. They can be tested by calling the **feof()** and **ferror()** functions, respectively. These flags can be manually cleared by calling the **clearerr()** function.

6.6 Restarting Tasks That Use pREPC+

It is possible to restart a task that uses pREPC+. Since pREPC+ can execute with preemption enabled, it is possible to issue a restart to a task while it is in pREPC+ code. Note that the **t_restart** operation does not release any memory, close any files, or reset **errno** to zero. If you wish to have **clean_ups**, then have the task check for restarts and do them as it begins execution again. However, once the task has been created, the value of **errno** is never reset to zero.

6.7 Deleting Tasks That Use pREPC+

To avoid permanent loss of pREPC+ resources, pSOS+ does not allow deletion of a task which is holding any pREPC+ resource. Instead, **t_delete** returns error code 0x19 which indicates the task to be deleted holds pREPC+ resources.

The exact conditions under which pREPC+ holds resources are complex. In general, any task that has made a pREPC+ service call may hold pREPC+ resources. **fclose(0)**, which returns all pREPC+ resources held by the calling task, should be called by the task to be deleted prior to calling **t_delete**.

pNA+ and pHILE+ also hold resources that must be returned before a task can be deleted. These resources are returned by calling **close(0)** and **close_f(0)** respectively. Since pREPC+ calls pHILE+, and pREPC+ calls pNA+ (if NFS is in use), these services must be called in the correct order. Below is a sample code fragment which a task can use to delete itself:

```
fclose(0));   /* return pREPC resources */
close_f(0);   /* return pHILE resources */
close(0);     /* return pNA resources */
t_delete(0); /* and commit suicide */
```

Obviously, close calls to components not in use should be omitted.

Since only the task to be deleted can make the necessary close calls, the simplest way to delete a task is to restart the task and pass arguments requesting self deletion. Of course, the task being deleted must contain code to handle this condition.

6.8 Deleting Tasks With exit( ) or abort( )

The **exit()** and **abort()** calls are implemented in pREPC+ as macros that are defined in the header file **prepc.h**. These macros, which the user needs to modify depending on which components are prepsent in the system, can be used to return all system resources and delete the task.

(Blank Page)

7 I/O System

A real-time system's most time-critical area tends to be I/O. Therefore, a device driver should be customized and crafted to optimize throughput and response. A driver should not have to be designed to meet the specifications of any externally imposed, generalized, or performance-robbing protocols.

In keeping with this concept, pSOS+ does not impose any restrictions on the construction or operation of an I/O device driver. A driver can choose among the rich set of pSOS+ system services, to implement queueing, waiting, wakeup, buffering and other mechanisms, in a way that best fits the particular driver's data and control characteristics.

The pSOS+ kernel includes an I/O supervisor whose purpose is to furnish a device-independent, standard method both for integrating drivers into the system and for calling these drivers from the user's application. I/O can be done completely outside of pSOS+. For instance, an application may elect to request and service some or all I/O directly from tasks. We recommend, however, that device drivers be incorporated under the pSOS+ I/O supervisor. pREPC+ and pHILE+ drivers are always called via the I/O supervisor.

7.1 I/O System Overview

Figure 17 illustrates the relationship between a device driver, the pSOS+ I/O system, and tasks using I/O services.

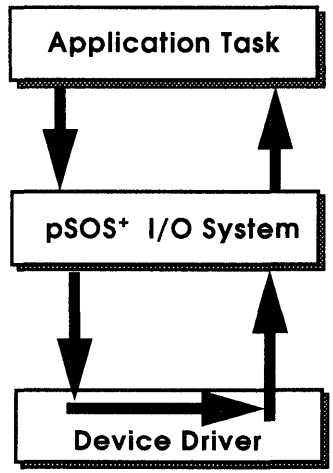

Figure 17. I/O System Organization

As shown, an I/O operation begins when an application task calls the pSOS+ I/O system. pSOS+ examines the call parameters and passes control to the appropriate device driver. The device driver performs the requested I/O service and then returns control to pSOS+, which in turn returns control back to the calling task.

Since device drivers are hardware dependent, the exact services offered by a device driver are determined by the driver implementation. However, pSOS+ defines a standard set of six I/O services that a device driver may support. These services are **de_init()**, **de_open()**, **de_close()**, **de_read()**, **de_write()**, and **de_cntrl()**. A driver may support any or all six of these services, depending on the driver design.

pSOS+ does not impose any restrictions or make any assumptions about the services provided by the driver. However, in general, the following conventions apply:

de_init() is normally called once from the **ROOT** task to initialize the device. It should be called before any other I/O services are directed to the driver.

de_read() and **de_write()** perform the obvious functions.

de_open() and **de_close()** are used for duties that are not directly related to data transfer or device operations. For example, a device driver may use **de_open()** and **de_close()** to enforce exclusive use of the device spanning several read and/or write operations.

de_cntrl() is dependent on the device. It may include anything that cannot be categorized under the other five I/O services. **de_cntrl()** may be used to perform multiple sub-functions, both input and output. If a device does not require any special functions, then this service can be null.

Note that the pSOS+ I/O system has two interfaces -- one to the application, the second to the device drivers. These two interfaces are described in more detail later in this chapter. First, it is helpful to introduce the I/O Switch Table.

7.2 I/O Switch Table

pSOS+ calls device drivers by using the I/O switch table. The I/O switch table is a user-supplied table that contains pointers to device driver entry points. The pSOS+ configuration table entries, **KC_IOJTABLE** and **KC_NIO** describe the I/O switch table. **KC_IOJTABLE** points to the table and **KC_NIO** defines the *number of drivers* in the table.

The I/O switch table is a linear array of pointers. Each contiguous group of eight pointers corresponds to one driver. Within each group of eight, the first six point to the device driver's entry points for the services **de_init()**, **de_open()**, **de_close()**, **de_read()**, **de_write()**, and **de_cntrl()**, respectively. The last two are reserved spares.

The location of a driver's pointers within the I/O switch table determine the *major device number* associated with the driver. The first eight entries correspond to major device 0, the second eight entries correspond to major device 1, and so on.

As an example, consider a system with 2 devices. In this case, the I/O switch table would consist of 16 entries and **KC_NIO** would be 2. Figure 18 illustrates the I/O Switch table structure for this example.

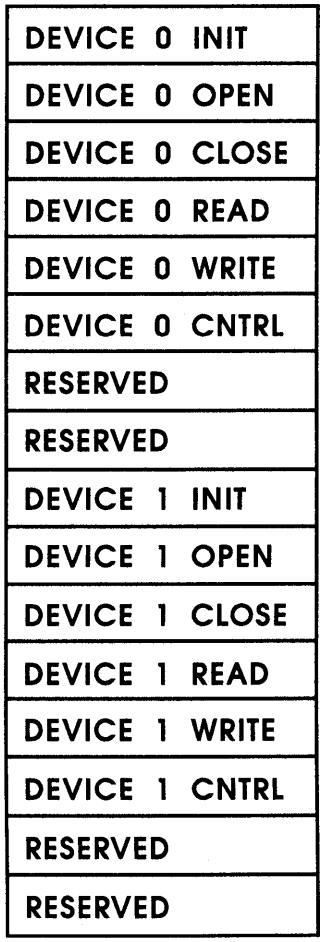

Figure 18. Sample I/O Switch Table

7.3 Application-to-pSOS+ Interface

The Application-to-pSOS+ Interface is defined by the following six system calls: **de_init()**, **de_open()**, **de_close()**, **de_read()**, **de_write()**, and **de_cntrl()**. The calling convention for each is as follows:

```
err_code = de_init(dev, iopb, &retval, &data_area)
err_code = de_open(dev, iopb, &retval)
err_code = de_close(dev, iopb, &retval)
err_code = de_read(dev, iopb, &retval)
err_code = de_write(dev, iopb, &retval)
err_code = de_cntrl(dev, iopb, &retval)
```

The first parameter, **dev**, is a 32-bit device number that selects a specific device. The most significant 16-bits of the device number is the major device number, which is used by pSOS+ to route control to the proper driver. The least significant 16 bits is the *minor device number*, which is ignored by pSOS+ and simply passed to the driver. The minor device number is used to select among several units serviced by one driver, but most drivers support only one unit and therefore also ignore it.

The second parameter, **iopb**, is the address of an I/O parameter block. This structure is used to exchange device-specific input and output parameters between the calling task and the driver. The length and contents of this I/O parameter block are driver specific.

The third parameter, **retval**, is the address of a variable that receives an optional quick-reference return value from the driver; for example, a byte count on a read operation. Use of **retval** by the driver is optional since values can always be returned via **iopb**. However, using **retval** is normally more convenient when only a single scalar value need be returned.

de_init() takes a fourth parameter, **data_area**. This parameter is no longer used, but remains for compatibility with older drivers and/or pSOS+ application code.

Each service call returns a function value that is zero if the operation is successful or non-zero if an error occurred.

Note that although pSOS+ does not define them, error codes below 0x10000 are reserved for use by pSOSystem components and should not be used.

With the following exceptions, error codes are driver specific:

- If the entry in the I/O Switch Table called by pSOS+ is -1, then pSOS+ returns a value of **ERR_NODR**, indicating that the requested service is not supported.
- If an illegal major device number is input, pSOS+ returns **ERR_IODN**.

Finally, note that if a switch table entry is null, pSOS+ returns 0.

7.4 pSOS+-to-Driver Interface

pSOS+ calls a device driver using the following syntax:

xxxxFunction(**ioparms**);

xxxxFunction is the driver entry point for the corresponding service called by the application. By convention, *Function* is the service name, while *xxxx* identifies the driver being called. For example, a console driver might consist of six functiuons called **CnslInit**, **CnslOpen**, **CnslRead**, **CnslWrite**, **CnslClose**, and **CnslCntrl**. Of course, this is just a convention -- any names can be used, since both the driver and the I/O switch table are user provided. Figure 19 illustrates this relationship.

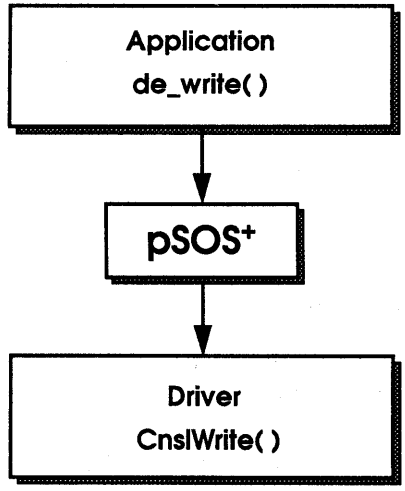

Figure 19. pSOS+-to-Driver Relationship

ioparms is a structure used to pass input and output parameters between pSOS+ and the driver. It is defined as follows:

```
struct  ioparms
{
  unsigned long used; /* Set by driver if this interface used */
  unsigned long tid;          /* task ID of calling task */
  unsigned long in_dev;       /* Input device number */
  unsigned long status;       /* Processor status of caller */
  void *in_iopb;          /* Input pnter to IO parameter block */
  void *io_data_area;         /* no longer used */
  unsigned long err;          /* For error return */
  unsigned long out_retval; /* For return value */
};
```

On entry to the driver, **used** is set to zero by pSOS+ and must be set to a non-zero value by the driver. It is used internally by pSOS+ when it receives control back from the driver.

> **CAUTION: It is imperative that your driver set the parameter "used" to a non-zero value, or else improper operation will result.**

On entry to the driver, **tid** contains the task ID of the calling task. It should not be changed by the driver.

On entry to the driver, **in_dev** contains **dev** as provided by the calling task; that is, the 32-bit device number. It should not be changed by the driver.

On entry to the driver, **in_iopb** points to the **iopb** provided by the calling task. It should not be changed by the driver.

On entry to the driver, the lower 16 bits of **status** contains the calling task's status register (SR) value prior to calling the pSOS+ I/O supervisor. **status** is normally not used.

io_data_area is no longer used.

err is used by the driver to return an error code, or 0 if the operation was successful. See section 7.3 for a discussion on error codes.

out_retval is used by the driver to return a quick-reference value to the calling task's **retval** variable. The contents of **out_retval** is copied into the variable pointed to by the service call input parameter **retval**.

7.5 Device Driver Execution Environment

Logically, a device driver executes as a subroutine to the calling task. Note, however, that on entry to pSOS+, the CPU enters the supervisor state. Thus, a device driver always executes in the supervisor state. This transformation can be useful on systems where the hardware restricts I/O address decoding to the supervisor state only.

Other characteristics of a task's mode remain unchanged by calling a device driver. Therefore, if a task is preemptible prior to calling a device driver, it remains preemptible while executing the driver. If a driver wants to disable preemption, it should use **t_mode()** to do so, being careful to restore the task's original mode before exiting. Similar caveats apply to Asynchronous Service Routines (ASRs).

Since a device driver executes as a subroutine to the calling task, it can use any pSOS+ system call. The following system services are commonly used by drivers:

Function	System Call
Waiting	**q_receive(), ev_receive(), sm_p()**
Wakeup	**q_send(), ev_send(), sm_v()**
Queueing	**q_receive(), q_send()**
Timing	**tm_tick()**, Timeout parameters on Waits
Mutual exclusion	**sm_p(), sm_v()**
Buffer management	**pt_getbuf(), pt_retbuf()**
Storage allocation	**rn_create()**

In addition, a device driver usually has an ISR, which performs wakeup, queueing, and buffer management functions. For a complete list of system calls allowed from an ISR, see Chapter 2, "pSOS+ Real-Time Kernel."

Note the following caveats regarding driver usage:

1. Device drivers execute using the calling task's supervisor stack. You must account for device driver stack usage when determining the stack sizes for tasks that perform I/O.
2. I/O calls can never be made from the pSOS+ task creation, task deletion, or context switch callouts.

3. I/O calls can never be made from an ISR.
4. In multiprocessor systems, I/O service calls can only be directed at the local node. pSOS+ does not support remote I/O calls. However, it is possible to implement remote I/O services as part of your application design; for example, with server tasks and standard pSOS+ system services.
5. Unlike other pSOS+ calls, I/O service calls do not automatically preserve all registers. While rarely a problem, refer to the *pSOSystem Programmer's Reference* for information on register usage by the I/O subsystem.

7.6 pREPC+ Drivers

As described in Chapter 6, "pREPC+ ANSI C Library," pREPC+ I/O can be directed to either disk files or physical devices. Disk file I/O is always routed via pHILE+, while device I/O goes directly to the pSOS+ I/O Supervisor. An I/O device driver that is called by pREPC+ directly via pSOS+ is called a *pREPC+ driver*, while a disk driver is called a *pHILE+ driver*, as illustrated in Figure 20.

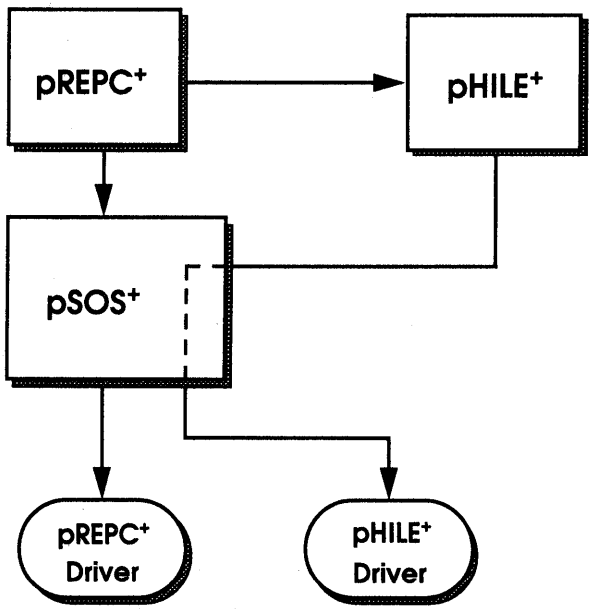

Figure 20. pHILE+ and pREPC+ Drivers

This section discusses pREPC+ drivers; section 7.7 covers pHILE+ drivers.

pREPC+ uses four pSOS+ I/O calls: **de_open()**, **de_close()**, **de_read()**, and **de_write()**. Therefore, a pREPC+ driver must supply four corresponding functions, e.g. *xxxx***Open()**, *xxxx***Close()**, *xxxx***Read()**, *xxxx***Write()**.

pREPC+ calls **de_open()** and **de_close()** when **fopen()** and **fclose()** are called, respectively, by your application. The corresponding driver functions that are called, *xxxx***Open()** and *xxxx***Close()**, are device specific. However, in general, *xxxx***Open()** will initialize a device, while *xxxx***Close()** will terminate I/O operations, such as flushing buffer contents. For many devices, these two routines may be null routines. pREPC+ does not pass an IOPB when calling **de_open()** and **de_close()**.

pREPC+ calls **de_read()** and **de_write()** to transfer data to or from a device. The I/O parameter block (IOPB) looks like the following:

```
typedef struct
{
     unsigned long count; /* no of bytes to read or write */
     void *address;       /* addr. of pREPC+ data buffer */
} iopb;
```

Recall that the IOPB is pointed to by the **in_iopb** member of the **ioparms** structure passed to the driver. **de_write()** results in a call to the driver function *xxxx***Write()**, which must transfer **count** bytes from the pREPC+ data buffer pointed to by **address**.

de_read() causes *xxxx***Read()** to be invoked, which transfers **count** bytes from the device to the pREPC+ buffer. *xxxx***Read()** is usually coded so that characters are read until a delimiter is detected or count bytes are received. Also, a pREPC+ *xxxx***Read()** driver routine usually implements backspace, line-erase and other line editing facilities.

*xxxx***Read()** and *xxxx***Write()** must return the number of bytes successfully read or written.

7.7 pHILE+ Drivers

Except for NFS volumes, pHILE+ accesses a volume by calling a device driver via the pSOS+ I/O supervisor. A driver invoked by pHILE+ is called a *pHILE+* driver.

When pHILE+ needs to read or write data, it calls the driver corresponding to the major/minor device number specified when the volume was mounted. pHILE+ uses only two of the six standard I/O system calls, **de_read()** and **de_write()**. Therefore, a pHILE+ driver only has to supply two functions, *xxxx***Read()** and *xxxx***Write()**. In practice, most pHILE+ drivers also provide an *xxxx***Init()** service, even though it is not called by pHILE+. It must be called independently by your application [via **de_init()**] prior to mounting the volume corresponding to the device. Similarly, even though **de_open()**, **de_close()**, and **de_cntrl()** are not used by pHILE+, a driver can implement these operations for physical I/O, error sensing, formatting, and so forth.

Like all drivers called by the pSOS+ I/O supervisor, pHILE+ drivers receive an **ioparms** parameter on input. Before a pHILE+ driver exits, it must store an error code indicating the success or failure of the call in **ioparms.err**. A value of zero indicates the call was successful. Any other value indicates an error condition. In this case, pHILE+ aborts the current operation and returns the error code back to the calling application. Error code values are driver defined. Any value can be used except values less than 0x10000, which are reserved.

7.7.1 The Buffer Header

When dealing with pHILE+ drivers, the IOPB parameter block pointed to by **ioparms.in_iopb** is called a *buffer header*. A buffer header has the following structure:

```
typedef struct buffer_header
{
    unsigned long b_device;   /* device major/minor number */
    unsigned long b_blockno;  /* starting block number */
    unsigned short b_flags;   /* block_type: data or control */
    unsigned short b_bcount;  /* number of blocks to transfer */
    unsigned long b_devforw;  /* system use only */
    unsigned long b_devback;  /* system use only */
    unsigned long b_avlforw;  /* system use only */
    unsigned long b_avlback;  /* system use only */
    void *b_bufptr;           /* address of data buffer */
    unsigned long b_bufwaitf;/* system use only */
    unsigned long b_bufwaitb;/* system use only */
    void *b_volptr;           /* system use only */
};
```

A pHILE+ driver uses only four of the parameters in the buffer header. They are the following:

b_blockno specifies the starting block number to read or write. **b_bcount** specifies the number of consecutive blocks to read or write. For more information on these parameters see section 7.7.3.

b_bufptr supplies the address of a data area; it is either the address of a pHILE+ cache buffer or a user data area. During a read operation, data is transferred from the device to this data area. Data flows in the opposite direction during a write operation.

b_flags contains a number of flags, most of which are for system use only. However, the low-order two bits of this field indicate the block type, as follows:

Bit 1	Bit 0	Explanation
0	0	Unknown block type
0	1	Data block
1	0	Control block

b_flags is used by more sophisticated drivers that take special action when control blocks are read or written. Most drivers will ignore **b_flags**.

b_flags low bits = 00 (unknown type) can occur only when **read_vol()** or **write_vol()** is issued on a volume that was initialized with intermixed control and data blocks. In this case, pHILE+ will be unable to determine the block type. If **read_vol()** or **write_vol()** is used to transfer a group of blocks that cross a control block/ data block boundary, these bits will indicate the type of the first block.

The remaining fields are for system use only.

The contents of the buffer header should not be modified by a driver. It is strictly a read-only data structure.

7.7.2 I/O Transaction Sequencing

pHILE+ drivers must execute transaction (i.e. read and write) requests that refer to common physical blocks in the order in which they are received. For example, if a request to write blocks 3-7 comes before a request to read blocks 7-10, then, because both requests involve block 7, the first request must be executed first.

If a pSOS+ semaphore is used to control access to a driver, then that semaphore must be created with FIFO queuing of tasks. Otherwise, requests posted to the driver might not be processed in the order in which they arrive.

7.7.3 Logical-to-Physical Block Translation

The **b_blockno** and **b_count** parameters together specify a sequence of logical blocks that must be read or written by the driver. However, most physical devices are not organized as a linear sequence of blocks. They are divided into sectors, tracks, cylinders, heads, and so forth. A pHILE+ driver must therefore translate "logical" block numbers provided by pHILE+ into "physical" block addresses on the device. How this is done depends on the type of device being accessed.

7.7.3.1 pHILE+ Formatted Volumes

For pHILE+ formatted volumes, a driver may implement any translation scheme that maps each logical block to a unique physical block. However, pHILE+ operates at maximum efficiency if blocks that are logically contiguous are also physically contiguous. Because of track to track transitions and other such boundaries, this usually is not entirely feasible, but a pHILE+ driver should minimize discontinuities.

7.7.3.2 MS-DOS Floppy Disk Format

For MS-DOS volumes, a driver must implement the same mapping used by MS-DOS; otherwise, your diskette will not be MS-DOS compatible. This section describes the required block mapping for each of the five MS-DOS floppy disk formats.

MS-DOS floppy disks have two sides, side 0 and side 1. On each side there are T tracks, numbered 0 to T-1. Each track contains S sectors numbered 1 to S. A sector is 512 bytes and maps directly to a pHILE+ block. A diskette thus contains (2 * T * S) sectors. The characteristics of each MS-DOS diskette are shown in Table 2.

Table 2. Characteristics of MS-DOS Diskettes

Capacity	Track Number	Sectors per Track
360 Kbyte	40	9
1.2 Mbyte	80	15
720 Kbyte	80	9
1.4 Mbyte	80	18
2.8 Mbyte	80	36

A block is mapped to a sector (head, track, sector) by the following rules:

1. The track number is first determined by dividing the block number by (2* *S*). The remainder, *R1*, is saved for Step 2.
2. *R1* is divided by *S* to obtain the side, 0 or 1. The remainder, *R2*, is saved for Step 3.
3. One is added to *R2* to obtain the sector number.

These rules are summarized by the following equations:

Track = Block / (2 * *S*) (remainder = *R1*)

Side = *R1* / *S* (remainder = *R2*)

Sector = *R2* + 1

An example:

On a 360-Kbyte diskette, T = 40 and S = 9. Block 425 is mapped as follows:

Track = 425 / (2 * 9) = 23 (remainder 11)

Side = 11 / 9 = 1 (remainder 2)

Sector = 2 + 1 = 3.

Thus, on a 360-Kbyte floppy, logical block 425 maps to:

Side = 1

Track = 23

Sector = 3

7.7.3.3 MS-DOS Hard Disk Format

The following equations apply to hard disks:

Cylinder = block/(sectors-per-track * heads)

Head = (block/sectors-per-track) MOD heads

Sector = (block-((block/sectors-per-track)*sectors-per-track))+1

Under pHILE+, an MS-DOS volume can be larger than 32 Mbytes. Due to an MS-DOS limit, the number of clusters in a volume can be only up to 65,535. To support volumes larger than 32M, cluster size should be larger than 512 bytes. Larger cluster size can cause inefficient use of disk space. To avoid this, a hard disk drive can be logically divided into partitions.

Each partition is used to hold one file volume. Hence, a partition can be either a DOS or pHILE+ volume. Partitioning allows heterogeneous file volumes to share a single drive. With partitions, multiple DOS volumes can be generated to cover large disk drives.

When a single hard disk drive contains multiple partitions, your driver must read the partition table (located in the master boot sector) during initialization and use the information in the table to translate sector addresses. Your application code and driver should use the upper byte of the minor device number to encode the partition number. Partition 0 should refer to the entire volume without partition translation. This convention allows pHILE+ and your application code to read any sector on the disk, including the master boot sector. Information about the encoding of partition numbers is explained in section 7.7.4.

7.7.4 MS-DOS Hard Drive Considerations

This section describes special considerations required when using MS-DOS hard drives with pHILE+.

You must provide a driver that performs the following functions in addition to the functions required by an MS-DOS floppy driver:

1. Logical-to-physical block translation. This function is required if the you want to create pHILE+ volumes in MS-DOS partitions, or MS-DOS volumes in drives with different than 512-byte sector size.
2. Partition table block translation.

Since pHILE+ always uses a logical block size of 512 bytes when accessing an MS-DOS volume, your driver need not perform logical-to-physical block translations if your drive's sector size is 512 bytes.

Your driver performs partition table block translation by reading the MS-DOS partition table from a drive's master boot sector during device initialization. Your driver stores this information in an internal table and uses it to translate a block address relative to the start of a partition into an absolute block address relative to the start of the drive. For example, if pHILE+ passes your driver a block address of 1000 and the desired partition starts at absolute block number 15500, the translated block address is 15500 + 1000 = 16500.

Although minor device number usage is transparent to pHILE+, you should use the following encoding scheme: the upper eight bits of the minor number contain a partition number, and the lower eight bits contain a drive number. MS-DOS allows a maximum of four partitions on any one drive. If this scheme is implemented, the following table maps minor numbers to physical drive number and partition.

Minor Number		Drive	Partition
256	(100H)	0	1
512	(200H)	0	2
768	(300H)	0	3
1024	(400H)	0	4
257	(101H)	1	1
513	(201H)	1	2
769	(301H)	1	3
1025	(401H)	1	4

Table 3. Minor Number to Drive/Partition Mapping

Since it is necessary for pHILE+ to read the partition table in the absolute sector 0 on an MS-DOS disk, your driver should recognize partition 0 as a partition spanning the entire disk; that is, your driver should not perform partition table translation on accesses in partition 0.

Assuming your driver follows these guidelines, prepare and make use of DOS hard drives in the pHILE+ environment, as follows:

1. Your drives must be low-level formatted following the instructions provided with your system. In most cases, your drives have already been formatted by the factory.
2. Your drives must be partitioned using the MS-DOS FDISK utility or other comparable utilities provided by some SCSI controller board venders. FDISK creates a partition table on the drive and allocates space for up to four partitions. You can use these four partitions to create any combination of MS-DOS and pHILE+ volumes. The total number of partitions can range from one to four.
3. Partitions that you intend to mount as MS-DOS volumes (using the pHILE+ **pcmount_vol** command) must also be formatted with the MS-DOS FORMAT command. Since FORMAT performs a high-level format on the partition, it is not necessary to use the pHILE+ **pcinit_vol** system call.

Although FDISK can only create "DOS" partitions, any such partition can be used as a pHILE+ volume. Simply create a partition of the desired size using FDISK, and when your system is running pSOSystem, initialize it using the pHILE+ **init_vol** system call. Be sure to record the number of blocks allocated to the partition; this information is needed when you use **init_vol** to initialize the volume. It is not necessary to format pHILE+ volumes using the MS-DOS FORMAT command. You can now mount the volume using the pHILE+ **mount_vol** system call.

7.8 Mutual Exclusion

If a device may be used by more than one task, then its device driver must provide some mechanism to ensure that no more than one task at a time will use it. When the device is in use, any task requesting its service must be made to wait.

This exclusion and wait mechanism may be implemented using a message queue or semaphore. In the case of semaphores, the driver's **init()** service would call **sm_create()** to create a semaphore, and set an initial count, typically 1. This semaphore represents a resource token. To request a device service, say **de_read()**, a task must first acquire the semaphore using the system call **sm_p()** with **SM_WAIT** attribute. If the semaphore is available, then so is the device. Otherwise, pSOS+ puts the

task into the semaphore wait queue. When a task is done with the device, it must return the semaphore using **sm_v()**. If another task is already waiting, then it gets the semaphore, and therefore the device.

In summary, a shared device may be protected by bracketing its operations with **sm_p()** and **sm_v()** system calls. Where should these calls take place? The two possibilities, referred to later as Type 1 and Type 2, are as follows:

1. **sm_p()** is put at the front of the read and write operation, and **sm_v()** at the end.
2. **sm_p()** is put in **de_open()**, and **sm_v()** in **de_close()**. To read or write, a task must first open the device. When it is finished using the device, the device must be closed.

Type 2 allows a task to own a device across multiple read/write operations, whereas with Type 1 a task may lose control of the device after each operation.

In a real-time application, most devices are not shared, and therefore do not require mutual exclusion. Even for devices that are shared, Type 1 is usually sufficient.

7.9 I/O Models

Two fundamental methods of servicing I/O requests are known; they are termed synchronous and asynchronous. Synchronous I/O blocks the calling task until the I/O transaction is completed, so that the I/O overlaps with the execution of other tasks. Asynchronous I/O does not block the calling task, thus allowing I/O to overlap with this, as well as other tasks. pSOS+ supports both methods.

The following sections present models of synchronous and asynchronous device drivers. The models are highly simplified and do not address hardware-related considerations.

7.9.1 Synchronous I/O

A synchronous driver can be implemented using one semaphore. If it is needed, Type 1 mutual exclusion would require a second semaphore. To avoid confusion, mutual exclusion is left out of the following discussion.

The device's **init()** service creates a semaphore **rdy** with initial count of 0. When a task calls **read()** or **write()**, the driver starts the I/O

transaction, and then uses **sm_p()** to wait for the **rdy** semaphore. When the I/O completion interrupt occurs, the device's ISR uses **sm_v()** to return the semaphore **rdy**, thereby waking up the waiting task. When the task resumes in **read()** or **write()**, it checks the device status and so forth for any error conditions, and then returns. This is shown as pseudo code below:

```
SYNC_OP:                        DEV_ISR:
    Begin                       Begin
      startio;                    transfer data/status;
      sm_p (rdy, wait);           sm_v (rdy);
      get status/data;          End;
    End;
```

An I/O transaction may of course trigger one or more interrupts. If the transaction involves a single data *unit*, or if the hardware provides DMA, then there will normally only be a single interrupt per transaction. Otherwise, the ISR will have to keep the data transfer going at successive device interrupts, until the transaction is done. Only at the last interrupt of a transaction does the ISR return the semaphore to wake up the waiting task.

7.9.2 Asynchronous I/O

Asynchronous I/O is generally more complex, especially when error recovery must be considered. The main advantage it has over synchronous I/O is that it allows the calling task to overlap execution with the I/O, potentially optimizing throughput on a task basis. The effect that this has at the system level is less clear, since multitasking ensures overlap even in the case of synchronous I/O, by giving the CPU to another task. For this reason, synchronous I/O should be used, unless special considerations require asynchronous implementation.

Note that if Type 1 mutual exclusion is required, it is normally taken care of by the asynchronous mechanism, without the need for extra code.

A simple, one-level asynchronous driver can be implemented using just one message queue. The device's **init()** service creates the queue **rdy** and sends one message to it. When a task calls **read()** or **write()**, the driver first calls **q_receive()** to get a message from the queue **rdy**, starts the I/O transaction, and then immediately returns.

The device's ISR, upon transaction completion, uses **q_send()** to post a message to the queue **rdy**. This indicates that the device is again ready. If this, or another, task calls the same device service before the last I/O transaction is done, then the **q_receive()** puts it into the wait queue, to wait until the ISR sends its completion message.

The pseudo code is as follows:

```
ASYNC_OP:                       DEV_ISR:
    Begin                       Begin
    q_receive (rdy, wait);      transfer data/status;
    startio;                    q_send (rdy);
    End;                        End;
```

This simplified implementation has two weaknesses. First, it does not provide a way for the device driver to return status information to more than one task. Second, at most only one task can overlap with this device. Once the device is busy, all requesting processes will be made to wait. Hence the term "one-level" asynchronous.

A more general and complex asynchronous mechanism requires one message queue and one flag, as follows. The device's **init()** service creates an empty message queue called **cmdq**. It also initializes a flag to **ready**.

The device's **read()** or **write()** service and ISR are shown below as pseudo code:

```
ASYNC_OP:                                   DEV_ISR:
Begin                                       Begin
  q_send (cmdq);                              cmd := q_receive (cmdq, no-wait);
  t_mode (no-preempt := on);                  if cmd = empty then
  if flag = ready then                           flag := ready;
    flag := busy;                             else
    cmd := q_receive (cmdq, no-wait);            flag := busy;
    if cmd = empty then                          startio (cmd);
         exit;                                endif;
    else                                    End;
        startio (cmd);
    endif;
  endif;
  t_mode (no-preempt := off);
End;
```

In essence, the queue **cmdq** serves as an I/O command queue for the device operation. Each command message should normally contain data or a buffer pointer, and also the address of a variable so that the ISR can return status information to a calling task (not shown in the pseudo code).

The **flag** global variable indicates whether the device is busy with an I/O transaction or not.

The **q_send()** system call is used to enqueue an I/O command. The **q_receive()** system call is used to dequeue the next I/O command.

The clause **cmd = empty** actually represents the test for **queue = empty**, as returned by **q_receive()**.

The **t_mode()** call is necessary to disable preemption, to prevent race on the **flag** variable.

There is no need to disable interrupts.

(Blank Page)

Index

E

F

G

H

I

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

(Blank Page)

Document Title: pSOSystem System Concepts
Document Number: PS2-000-003
Part Number: PS2000MAN
Revision Date: 6 December 1993

Corporate Headquarters

3260 Jay Street
Santa Clara, CA 95054
408-980-1500 phone
408-980-0400 fax
pSOS_sales@isi.com e-mail

International Offices

Integrated Systems, Inc., Ltd.
First Floor, Gate House
43 Fretherne Road
Welwyn Garden City
Hertfordshire AL8 6NS
England
+44-707-331199 phone
+44-707-391108 fax

Integrated Systems, S.A.
1 rue du Petit Robinson
78350 Jouy-en-Josas
France
+33-1-34-65-3732 phone
+33-1-34-65-0034 fax

ARS Integrated Systems
Starnberger Strasse 22
82131 Gauting, Munich
Germany
+49-89-850-6081 phone
+49-89-850-8918 fax

Integrated Systems, S.A.
Co Enator
Kronborgsgrand 1
S-16487 Kista
Sweden
+46-8-703-6200 phone
+46-8-703-6283 fax

Integrated Systems, Inc. AB
Sisjo Kullegata 8
S-421 32 Vastra Frolunda
Goteburg
Sweden
+46-31-683750 phone
+46-31-683332 fax

Integrated Systems, Inc.
Tokyo Branch Office
AHM'S 1
1-18-4 Ebisu Nishi
Shibuya-ku, Tokyo 150
Japan
+81-3-5489-0171 phone
+81-3-5489-0174 fax

U.S. Offices

AZ, Tempe
602-968-8280 phone
602-967-3956 fax

CA, Burbank
818-972-1747 phone
818-972-1601 fax

CA, El Segundo
310-364-5282 phone
310-364-5206 fax

CA, San Ramon
510-830-9667 phone
510-830-0950 fax

FL, Orlando
407-857-5790 phone
407-857-5740 fax

IL, Chicago
708-498-7380 phone
708-498-7384 fax

MA, Burlington
617-272-0773 phone
617-272-3270 fax

MA, Northborough
508-393-1231 phone
508-393-5194 fax

NC, Raleigh
919-846-7340 phone
919-676-7005 fax

NJ, Phillipsburg
908-454-8899 phone
908-454-8042 fax

OH, Cleveland
216-779-3255 phone
216-777-4205 fax

TX, Dallas
214-770-7882 phone
214-770-7885 fax

TX, Houston
713-831-6855 phone
713-831-6804 fax

Washington, D.C.
703-391-6027 phone
703-391-6004 fax

integrated systems