microware

Training and Education
Presents:

OS-9 Advanced Topics

microware

Process Scheduling Definitions

tick - the smallest measurable amount of time on the OS-9 system. By default, the OS-9 system is configured to receive 100 clock interrupts (ticks) per second. This tick rate may be changed provided that your hardware supports other values and you have source code to your system's clock device driver. If the tick rate is changed, the change does not take effect until the next reboot.

slice - this is an amount of time, measured in ticks. A slice is the amount of time a process may remain in the CPU (execute) before the kernel decides to perform a context switch. (Note that this is without regard to pre-emption, and that although a context switch takes place, sometimes the process leaving the CPU is also the process to run next.) By default there are two ticks per slice. This value may be changed on the fly by modifying a system global and may be changed on a quasi-permanent basis by modifying the init module used to boot with.

When a process is placed in the CPU to resume (or begin) execution, it will stay there for the number of ticks specified by the system global **d_tslice**. After this time expires, the kernel causes a context switch, allowing the next process to begin its "time slice".

Process Scheduling

OS-9 uses four methods of determining what process to run: round robin, priority, aging, and pre-emption. These scheduling mechanisms are all used in conjunction with one another.

Round robin: by itself, this form of process scheduling is a fair arbiter which basically says that every process gets an equal turn in the CPU, and when that turn expires, it goes to the end of the line of processes that want the CPU.

Priority: this form of scheduling, used by itself, requires that the process with the highest priority is allowed to run as long as it wants, and as long as it has the highest priority no other process can get any CPU time.

Aging: this is a mechanism that must work with priority. Aging says that when process A gets a time slice (due to its high priority), all other processes age by one. That is, their priorities artificially increase by one.

Pre-emption: this is the act of a non-running process forcing the running process from the CPU.

The Process Scheduling Function

Assume that all processes are eligible for the cpu at all times. When a process is created, it is placed within the active queue. The active queue is a line of processes that want CPU time. This queue is kept sorted in order of the next process to enter the CPU to the last. The process being placed into the queue may or may not be placed at the end of the queue; priority takes a role.

Every process that enters the active queue is assigned an age equal to the process' priority. The sort of the queue is thus based on each process' age rather than directly on its priority. The age of the process being inserted is compared with the ages of processes already in the queue, from the rear and moving forward. When the age of a process already in the queue is less than that of the process being inserted, the process being inserted moves forward. When the age of the process already in the queue is equal to the age of the process being inserted (or exceeds its age) the process being inserted is destined to fall behind this "equal or greater" process. When the kernel needs a process to execute, it takes the process in the front of the queue; this process has the highest (or, oldest) age.

When a process' time slice expires, it is given an age again equal to its priority, and shuffled into the active queue. Now, whatever process is in the front of the queue gets to run. (Yes, it may even be the same process.) All processes in the queue at the time of the context switch (that were not a part of the switch) are aged by one. (Their ages all increase by one.)

Process Scheduling Example

Assume there are three processes, A, B, and C, with respective priorities 130, 128, and 128. Also, assume they all enter the active queue in alphabetical order. Initially, there will be no process in the CPU, and the active queue will contain in order A, B, and C with respective ages at 130, 128, and 128. The system will have the following active queue layouts as time progresses:

Table 7: Process Scheduling Dynamics

CPU	Active Queue		
none	A_{130}	B_{128}	C_{128}
A	A_{130}	B_{129}	C_{129}
A	B_{130}	C_{130}	A_{130}
B	C_{131}	A_{131}	B_{128}
C	A_{132}	B_{129}	C_{128}
A	B_{130}	A_{130}	C_{129}
B	A_{131}	C_{130}	B_{128}
A	C_{131}	A_{130}	B_{129}

Process Queues

Processes generally do not spend much time in the active queue since very few processes actually want the CPU 100% of the time. (This was the assumption made in the previous examples.) For example, when a process sleeps, waits for child processes to terminate, or waits for favorable semaphores and events, they cannot use the CPU, so there are queues other than the active queue to place these processes.

Active queue: This queue holds the process descriptors of all processes that are ready to use the CPU. This list is sorted in order of the next process to enter the CPU to the last.

Sleep queue: This queue holds sleeping processes which may either be awakened through receiving a signal or sleeping their allotted time. It is kept sorted in order of the next process to be (naturally) awakened to the last, followed by processes that have no specific wakeup time.

Wait queue: This queue holds processes that are waiting for child processes to terminate. This is an unsorted list since the kernel cannot tell which processes will leave the queue first.

Preemption

Preemption is defined to be the act of one process forcing another out of the CPU, typically so that this preempting process can execute. Under OS-9, only a relatively high priority process can preempt a relatively low priority process. Preemption can only occur at the time the higher priority process enters the active queue. This process may enter the active queue either by receiving a signal, waking up from a timed sleep, or any other means of activation (such as being created.)

If a process is activated and, at this time, its priority does not allow it to preempt the current running process, the activated process will enter the active queue to be placed wherever its age (set equal to its priority) calls for with regards to other process in the active queue. Thus, a process receiving a signal gets no special privilege unless it can preempt the current running process. When a process does force another to prematurely leave the CPU, the preempting process has no guarantee of entering the CPU immediately; rather, its position within the active queue determines the next process to enter.

System Globals

The operating system kernel, just like any other program, maintains a set of global variables. Since the kernel's globals are accessible to all processes running on the system, these variables are called system globals. These system globals may be viewed by all processes, and, super user processes may change certain globals. (A super user process is any process owned by group zero.) We view all globals on the system by typing **vos globs** at the shell prompt.

Within the context of process scheduling, there are three globals of particular interest: **d_tslice**, **d_minpty**, and **d_maxage**.

d_tslice, as previously discussed, has a default value of two. This means that a process has two clock interrupts of execution time before the kernel can decide to perform a context switch.

d_minpty specifies the minimum priority a process must have in order to be scheduled for execution time. When this value is non-zero, the system will neither consider for execution nor age those processes with a priority below d_minpty. This is true even if there are no remaining processes with priorities equal to or greater than d_minpty!

d_maxage determines the highest value any process may age to. Setting d_maxage to one will implement a strict priority based system since no processes can age. When d_maxage is zero, a processes maximum age is 65535; when d_maxage is 100 a processes maximum age is 99. Note, however, that the lowest a process' age can ever be is equal to its priority, no matter how low d_maxage is.

Viewing and Changing System Globals

There are two C functions for dealing with system globals: **_os_getsys()** and **_os_setsys()**. To use these functions, it is advised that you include two header files: <sysglob.h> and <stddef.h>. (The former describes the structure of the system globals while the latter contains a macro, offsetof(), which allows us to get the offset to an element within a structure.)

```
error_code _os_getsys(u_int32 offset, u_int32 size, glob_buff
       *yourvar);
error_code _os_setsys(u_int32 offset, u_int32 size, glob_buff
       yourvar);
```

The program below demonstrates the viewing and changing of system globals.

```
#include <stdio.h>
#include <stddef.h>
#include <sysglob.h>
#include <types.h>
#include <errno.h>
```

```
main()
{
  glob_buff d_maxage;
  errno = _os_getsys(offsetof(sysglobs, d_maxage),
    sizeof(d_maxage.wrd),&d_maxage);
  if (errno != 0) {
    fprintf(stderr,″Error getting d_maxage!\n″);
    _os_exit(errno);
  }

  printf(“Current value of d_maxage: %d\n”,d_maxage.wrd);
  printf(“New value: ”);
  scanf(“%d”,&d_maxage.wrd);

  errno = _os_setsys(offsetof(sysglobs,d_maxage),
    sizeof(d_maxage.wrd),d_maxage);
  if (errno != 0) {
    fprintf(stderr,″Error setting d_maxage! Are you SUPER?\n″);
    _os_exit(errno);
  }

  printf(“It has been done. Good bye!\n”);
}
```

A few other globals of interest: **d_ticks**, **d_tcksec**.

Memory Management

Under OS-9, all processes share a common address space consisting of a single memory map. Virtual memory is not supported in any form as it would impose a large tax on the number of CPU cycles available for your programs. (Lack of virtual memory means we cannot perform "garbage collection" on memory in order to reduce apparent fragmentation.) Although we do not support virtual memory, we do provide some support for memory management hardware. The system extension **ssm** (system security module) utilizes an MMU to insure that a process can only access memory it has been granted. This improves security on a system as well as prevents most processes from causing harm to other processes or the system itself.

OS-9 memory does not have to be contiguous, but each block of memory does have to be known to the system at boot time. (The system learns of areas of memory by looking at the init module, whose source code describes the various areas.) Of course, if memory on the system is not contiguous, the system memory will begin to fragment.

The command **mfree** shows how much memory is currently available. This number is not very useful, however, as fragmentation is not addressed. The "-e" option to mfree changes the output to include all individual fragments of free memory as well as the amount of memory found at boot time.

Memory Allocation

There are a number of calls available for the allocation of memory. Some of the major calls are presented here.

memptr = malloc(size): The malloc() function is a general purpose memory allocation call. It is the defacto standard for memory allocation in C. This does not necessarily make it the best choice, but it is the most common choice. In general terms, malloc is much more than an allocation call, it is a hook into a mini allocation system. This function does not always extract memory from the system, since when it does get new memory from the system it generally gets more than is needed. This malloc system keeps detailed records of what memory has been received from the system but not yet allocated to the user, and often grants memory from this area. Memory granted through malloc() is returnable through the **free()** function.

When malloc does get memory from the system, it gets memory in increments of the system's minimum block size (4k by default.) A pointer to the number of bytes requested by the user is returned when this call succeeds, and a null pointer is returned when it fails. A related call, **_mallocmin()**, is used to change the number of bytes requested by malloc from the system. This change must still be a multiple of the system's minimum block size. Larger minimums can help reduce memory fragmentation at the expense of tying up more memory than is needed.

```
error_code _os9_srqmem(u_int32 *size, void **ptr);
```

This call allows you to allocate memory from the system in a manner requiring less overhead from the kernel than malloc().

```
error_code _os_srqmem(u_int32 *size, void **ptr, u_int32 color);
```

Similar to the call above, this call allows you to specify which memory region the memory should come from.

```
error_code _os_srtmem(u_int32 size, void *ptr);
```

This call allows you to return memory granted via one of the above calls.

```
error_code _os_mem(...);
```

Does not work! Only present for compile compatibility with previous versions of the operating system.

Subroutine Modules (OS-9)

The subroutine module is one of two methods provided for placing code in an external location. They provide a simple means of implementing a shared library which is dynamically linked. Since they are linked at run-time rather than compile time, they may never refer to a program's global variables by direct reference to their names. (Pointer access is fine though.) Also, a subroutine module is not permitted to have any global variables of its own.

Architecturally, a subroutine module closely resembles a program module. The primary difference is a subroutine module does not have an initial execution point, since the system does not call the module. (OS-9 has very little overhead, or control over subroutine modules. Therefore, anything goes.)

Since there is no required structure inside a subroutine module, they can be difficult to use. Microware does use a preferred structure, but there is no requirement to do so. Inside a subroutine module there are bound to be several externally callable routines. The biggest problem facing the program using this library is finding the routines! That program can, however, easily determine two things about the module: The base address (once it is in memory) and the address of the data area. If we adopt the standard of placing an index table at the location refered to as the data area, we can use that index to find the routines in the module.

C and Assembly

On occasion you will need to mix C and assembly language for one purpose of another. If this is in your future, you must understand the register convention used by Microware's C compiler for its own functions. This convention is shown below.

Table 8: C Register Convention

Register(s)	Purpose
d0, d1	First two function parameters. Remaining parameters are pushed onto the stack.
d0	Function return value.
d2, d3	Compiler temporaries for sub expressions et. al.
d4 - d7	Register class variables.
a0, a1	Compiler temporaries for pointer arrithmetic.
a2 - a4	Register class pointers.
a5	Stack frame pointer.
a6	Static storage pointer.

Trap Modules

Trap modules provide functionality similar to that of subroutine modules. There are several differences, but the primary difference is that subroutine modules require no overhead on the operating system or CPU (other than that required to execute the code, of course) and trap modules do. To call a routine in a subroutine module, the calling program simply jumps to the proper address. To call a routine in a trap module, the caller must cause a hardware exception.

The 68K processor provides 16 trap vectors with which to cause an exception; vector zero is reserved by the operating system kernel. This leaves 15 vectors remaining, and OS-9 allows each process to have its own set of up to 15 simultaneous trap handlers. These trap vectors get resolved within a trap module.

The architecture of a trap module resembles that of the suggested use of a subroutine module. One major difference, however, the trap module has exactly three entry points, and its index table has offsets to exactly two! Recall, the subroutine module pointed to an index table (of unknown length) through the _mexec offset. The trap module points to the third entry point via _mexec, while the index table is made of the first two long words inside the body of the module. (The kernel considers these fields an extended part of the header.) Common names for these threee entry points are TrapInit, TrapTerm (both in the index table) and TrapEnt.

A calling program makes the trap module available by installing it upon one of its trap vectors. Since trap vectors 13 and 15 may be used by the compiler, it is best to steer clear of these if possible. The compiler uses trap vector 13 for cio or csl, and an older version of the compiler used vector 15 for floating point math purposes. The function _os_tlink() is used to attatch to a trap module.

```
error_code _os_tlink(u_int32 vector, char *modname, void **trapent,
                  mh_trap **modptr, void *nothing, u_int32 xtra_mem);
```

The TrapInit entry point is called every time a process tlink's to a trap module. Trap modules are given static storage every time they are linked to; if the module needs any globals initialized, the TrapInit routine is the place to do it.

The TrapTerm routine is not currently used by the system, thus it has no specific responsibility.

When the calling program wishes to use the trap module, a hardware exception must be generated. The 68K "trap" instruction is followed with a word, between 0 and 15, which identifies the vector to generate the exception on. This vector must match the vector defined in the _os_tlink() call, 1 through 15.

Since you may wish to use a trap library for multiple routines, the word following the trap vector is passed to the TrapEnt routine when the trap instruction is called. One caveat: the trap instruction is not callable from C. However, Microware's assembler recognices the "tcall" macro which takes two word parameters, the vector and function word, to call the trap instruction appropriately. Your class directory contains a full trap example program.

Exception Handling

Under normal circumstances, when a process causes a hardware exception, the kernel kills the offending process. This treatment can be alleviated if the process first informs the kernel of a special exception handling routine for the exception(s) which may be caused. If this has been done, rather than kill the process, the kernel forces it to immediately execute its handling routine when it next gets the CPU.

The kernel is informed of a process' exception handling capabilities through the _os_strap() system call. The first parameter is a pointer to a special stack to use when in the handler(s) and the second is a pointer to a table used to initialize the handler(s).

```
#include <settrap.h>
#include <MACHINE/reg.h>
error_code _os_strap(u_int32 *stack, strap *init_tbl);
/* u_int32 div0func(u_int32 vect, u_int32 pc, u_int32 addr) */

strap init_tbl[] = {
   {T_ZERDIV,div0func},
   {-1,NULL}
};
```

Interrupt Service Routines

The OS-9 interrupt model includes the system kernel as a front end to your interrupt service routines. This is to allow multiple, independent interrupt service routines to be installed on a single vector.

OS-9 interrupt service routines come in two varieties: fast and shared. A fast ISR is guaranteed to be the first one called on a vector, and must save and restore all registers used except **d0** and **a2**. There may only be one fast IRQ per vector. Shared IRQs line up on their vector with an order dependant on their priorities. When the kernel is called for an interrupt request on a vector, the ISR list for that vector is traversed until one of the routines claims responsibility for the interrupt. Shared ISRs must preserve registers **d2 - d7**, **a1**, **a4**, **a5**, and **a7**.

```
_os_firq(u_int32 vector, 0, void *irq_rtn, void *statics);
_os9_irq(u_int32 vector, u_int32 priority, void *irq_rtn,
  void *glob_data, void *port_addr);
```

Process Scheduling

OS-9 is a multi-tasking operating system; that is, two or more independent programs, called processes or tasks, can be executed simultaneously. Several processes share each second of CPU time. Although the processes appear to run continuously, the CPU only executes one instruction at a time. The OS-9 kernel determines which process to run and for how long based on the priorities of the active processes. The action of switching from the execution of one process to another is called ***task-switching***. Task-switching does not affect the programs' execution.

The CPU is interrupted by a real-time clock every ***tick***. By default, a tick is .01 second (10 milliseconds). At any occurrence of a tick, OS-9 can suspend execution of one program and begin execution of another. The tick length is hardware dependent. To change the tick length, you must change the D_TckSec system global in the Init module, modify the clock driver, and re-initialize the hardware.

The longest amount of time a process will control the CPU before the kernel re-evaluates the active process queue is a called a ***slice*** or ***time-slice***. By default, a slice is two ticks. To change the number of ticks per time-slice, the standard method is to modify the Init module.

Process Creation

A process must be created before it can be scheduled for execution. New processes are created by the F$Fork system call. The most important parameter passed in the "fork" system call is the name of the primary module that the new process is to execute immediately. The following list outlines the creation process:

① **Locate or Load the Program.**
OS-9 tries to find the module in memory. If the module is already in memory, OS-9 goes on to the next step. If the module is not located in memory, OS-9 searches for the module in the following places:

a. The default execution directory
b. Directories specified by the PATH environment variable
c. The current data directory or an explicitly stated directory (for example, /h0/USR/TE1/DO.IT) are searched for a procedure file.

If the module is found, it is loaded into memory and OS-9 goes on to the next step. If it is not found, an error is returned.

② **Allocate and Initialize a Process Descriptor.**
After the primary module has been located, a data structure called a ***process descriptor*** is assigned to the new process. The process descriptor is a table containing information about the process: its state, memory allocation, priority, I/O paths, etc. The process descriptor is automatically initialized and maintained. The process need not be concerned about the descriptor's existence or contents.

③ **Allocate the Stack and Data Areas.**
The primary module's header contains a data and stack size. OS-9 allocates a ***contiguous memory area*** of the required size from the free memory space. **NOTE:** The malloc() and srqmem() calls can be used to allocate more memory. However, OS-9 only allows 32 discontinuous areas per process.

④ **Initialize the Process.**
The new process' registers are set to the proper addresses in the data area and object code module. If the program uses initialized variables and/or pointers, they are copied from the object code area to the proper addresses in the data area.

⑤ **Insert the Process in the Active Queue.**

Process Queues

Once created, a process can be in one of three states: active, waiting, or sleeping. Each state has an associated process ***queue***. A queue is a linked list of process descriptors. When a process moves from one queue to another, the process descriptor is moved from the current queue to the new queue.

Processes in the active queue are active and ready for execution. Active processes are given time for execution according to their relative priority with respect to all other active processes in the queue. All active processes receive some CPU time, even if they have a low relative priority. This will be discussed in the next section.

Wait Queue

A process in the wait queue is inactive until a child process terminates or a signal is received. The wait state is entered when a process executes an F$Wait system service request. The process remains inactive until one of its descendant processes terminates or the process receives a signal. Figure 1 illustrates how a process moves from the active queue to the wait queue and back again:

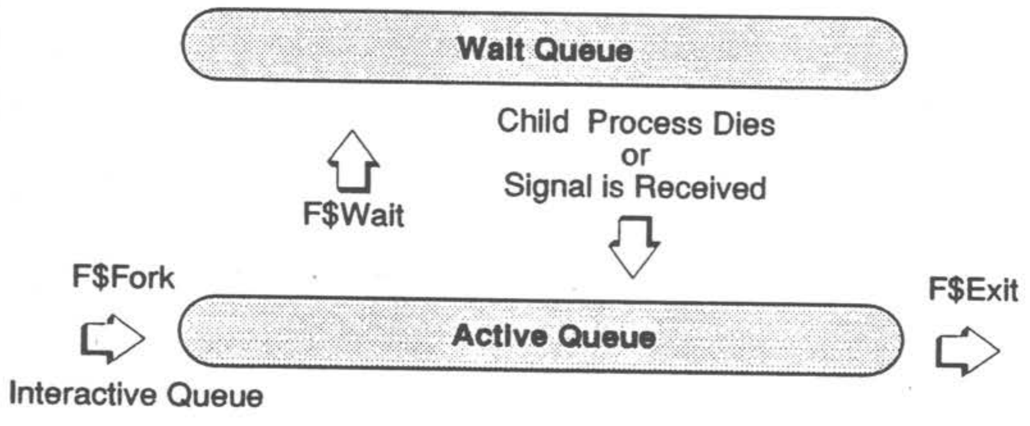


Figure 1: A process enters the active queue with an F$Fork system call. If an F$Wait system service request is received, the process is placed in the wait queue. Once in the wait queue, the process waits for one of two conditions to occur: either a child process dies or a signal is received. When one of the conditions is met, the process moves back into the active queue.

Sleep Queue

When a process is in the sleep queue, it is inactive for a specified time period or until a signal is received. The sleep state is entered when a process executes an F$Sleep service request. The F$Sleep request specifies a time interval for which the process is to remain inactive. Processes often sleep to avoid wasting CPU time while waiting for some external event, such as the completion of I/O. Zero ticks specifies an infinite period of time. Processes waiting on an event are also included in the sleep queue. Figure 2 illustrates how a process moves from the active queue to the sleep queue and back again:

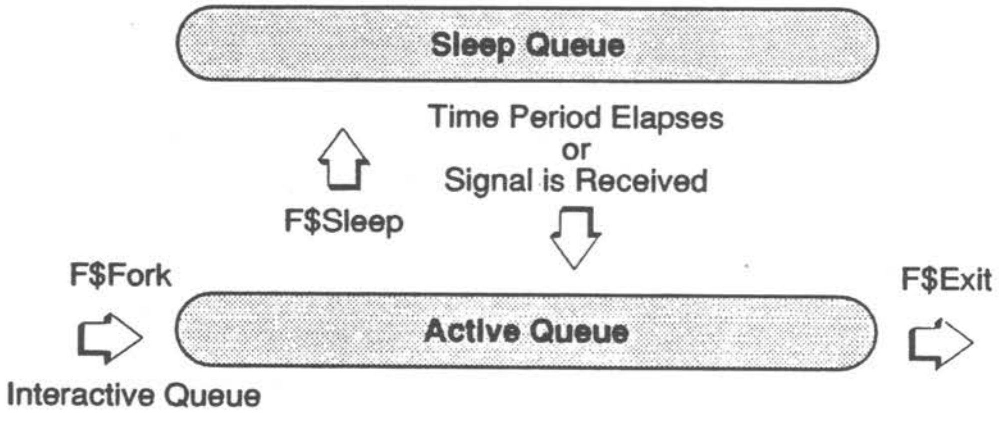


Figure 2: A process enters the active queue with the F$Fork system call. If an F$Sleep service request is received, the process is placed in the sleep queue. Once in the sleep queue, the process sleeps until one of two conditions occur: either the specified time period elapses or a signal is received. When one of the conditions is met, the process moves back into the active queue.

There are three different types of sleep using the tsleep() C call:

- tsleep(0);
  The process sleeps indefinitely until a signal is received.
- tsleep(1);
  The process gives up the remainder of its time-slice.
- tsleep(n);
  The process sleeps for n ticks.

OS-9's Scheduling Algorithm

To explain OS-9's scheduling algorithm, the individual concepts involved are discussed, starting with the most basic concept.

Round Robin

The round robin scheduling algorithm is the easiest scheduling algorithm to implement. With round robin, each process receives an equal share of the available CPU time. For example, if three processes are in the active queue, each process receives one-third of the CPU time. This algorithm is generally fair. However, if a process has a time-critical task, it may not be able to complete the task within the time restraints if too many other processes are running.

Priorities

Priorities allow some processes to run before others. For example, if two processes are in the active queue and one process has a higher priority than the other, the process with the higher priority receives the CPU time. When priorities are used with the round robin algorithm, the highest priority process runs until it waits, sleeps, or terminates. All equal priority processes share the CPU. This creates a problem because all low priority tasks stop while the higher priority tasks run. This is called ***starvation*** or ***suffocation***.

Aging

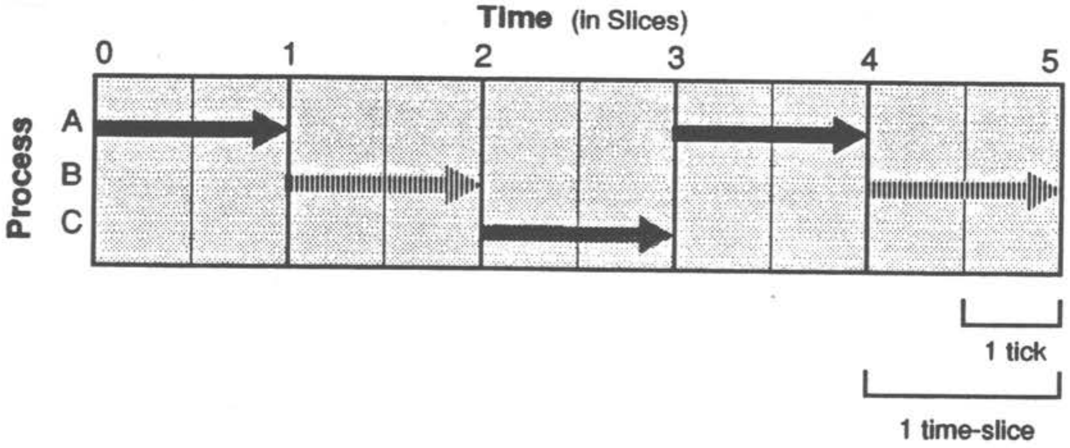
Aging allows lower priority tasks to eventually get some CPU time while a higher priority task is in the active queue. OS-9 assigns priorities on a scale from 1 to 65535, with one being the lowest. Processes enter the active queue with an age equal to their priority. The kernel schedules the process in the active queue with the highest age. There are four basic rules of aging:

- If a process in the active queue is not selected to receive CPU time, the process' age is incremented by 1.
- The process with the highest age receives CPU time.
- When a process finishes executing, it re-enters the active queue at an age equal to its initial priority.
- The active queue is always kept sorted by age.

Pre-Emptive Active Queue

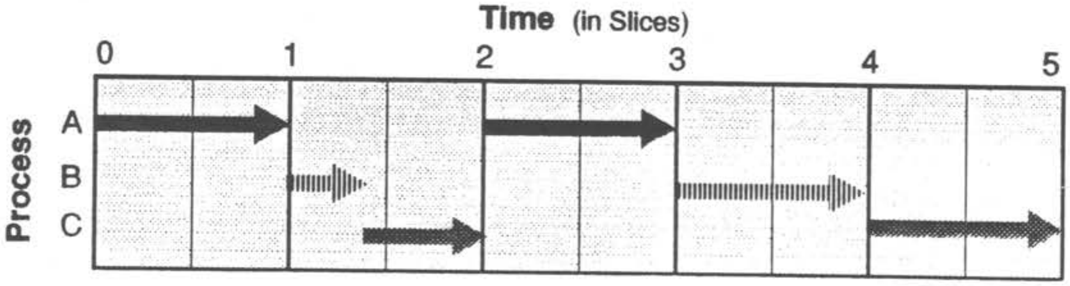
During critical real-time applications, fast response time is sometimes necessary. OS-9 provides this by pre-empting the currently executing process when a process with a higher priority becomes active. The lower priority process loses the remainder of its time-slice and is re-inserted into the active queue. The higher priority process is then given a full time-slice.

For example, when no signals are sent to processes in the wait or sleep queues, the processes executing in the CPU can be represented as the following:



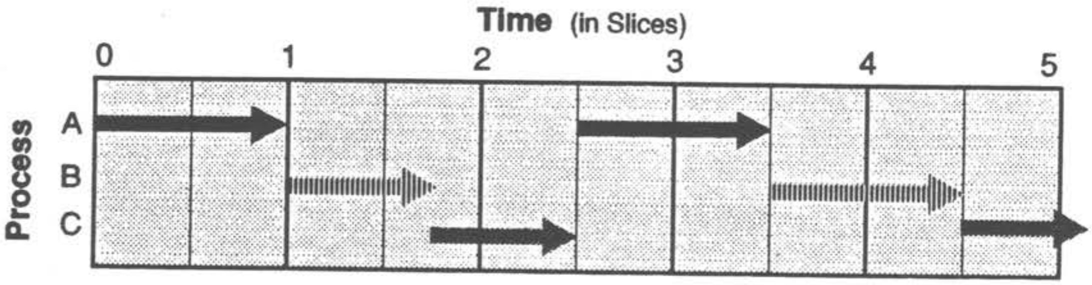
At the end of each tick, the CPU checks to see how long the current process has been executing. If the process has only been executing for one tick, the CPU allows the process to continue executing for a second tick. If the process has been in the CPU at the end of two ticks, the process is placed back in the active queue and a new process begins execution. In this example, each process enters the CPU, runs for a time-slice, and re-enters the active queue.

If process C is in the sleep (or wait) queue and receives a signal while process B is in its first tick and if process B has a lower priority, process C pre-empts process B. That is, process B gives up the remainder of its time-slice to process C:



Notice that process C only uses the remainder of process B's time-slice. Process C was executing at the end of the fourth and fifth ticks. Therefore, the CPU placed process C back in the active queue and began processing process A.

If process C received the interrupt during process B's second tick, process C would still pre-empt process B and run for two time-slices. However, at what would have been the end of process B's time-slice, the kernel would see that process C had not been executing at the end of two ticks and would allow it to continue executing for another tick:



D_MinPty and D_MaxAge

Task-switching is also affected by two system global variables: D_MinPty (minimum priority) and D_MaxAge (maximum age). Both variables are initially set in the Init module. Super users can access the variables through the F$SetSys system call.

D_MinPty defines a minimum priority below which processes are neither aged nor considered candidates for execution. Processes with priorities less than D_MinPty remain in the waiting queue and continue to hold any system resources that they held before D_MinPty was set.

D_MinPty is usually set to zero. All processes are eligible for aging and execution when this value is set to zero because all processes have an initial priority greater than zero.

Figure 3 illustrates D_MinPty.

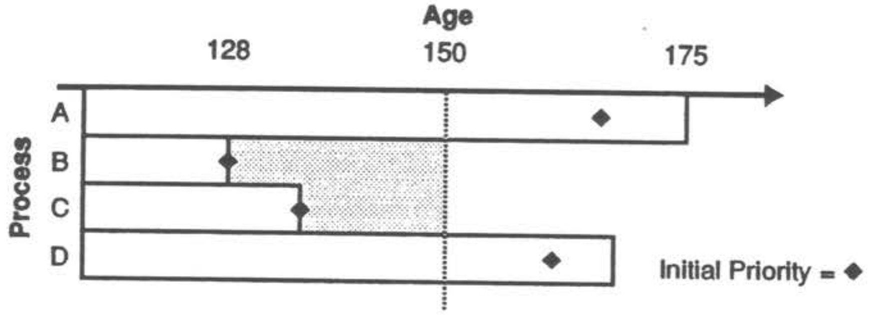


Figure 3: Four processes are in the active queue. Processes A and D have initial priorities greater than 150. Processes B and C have priorities less than 150. If D_MinPty is set at 150, only processes A and D will be eligible for aging and execution. Processes B and C will not be executed even if the other two processes sleep, wait, or terminate.

WARNING: D_MinPty is potentially dangerous. If the minimum system priority is set above the priority of all running tasks, the system completely shuts down and can only be recovered by a system reset. It is crucial to restore D_MinPty to zero when the critical task finishes or to reset D_MinPty or a process' priority in an interrupt service routine.

D_MaxAge defines a maximum age over which processes are not allowed to mature. By default, this value is set to zero. When D_MaxAge is set to zero, it has no effect on the processes waiting to use the CPU.

When set, D_MaxAge essentially divides tasks into two classes: low priority and high priority. A low priority task is considered to be any task with a priority below D_MaxAge. Low priority tasks continue aging until they reach the D_MaxAge cutoff, but they are not executed unless there are no high priority tasks waiting to use the CPU.

A high priority task is any task with a priority above D_MaxAge. A high priority task receives the entire available CPU time, but it is not aged. When the high priority tasks are inactive, the low priority tasks are run.

NOTE: If you wish to establish a strictly priority-based scheduling algorithm, set D_MaxAge to 1. Setting D_MaxAge to 1 effectively shuts off the aging mechanism. Therefore, aging does not affect the scheduling algorithm.

Figure 4 illustrates D_MaxAge.

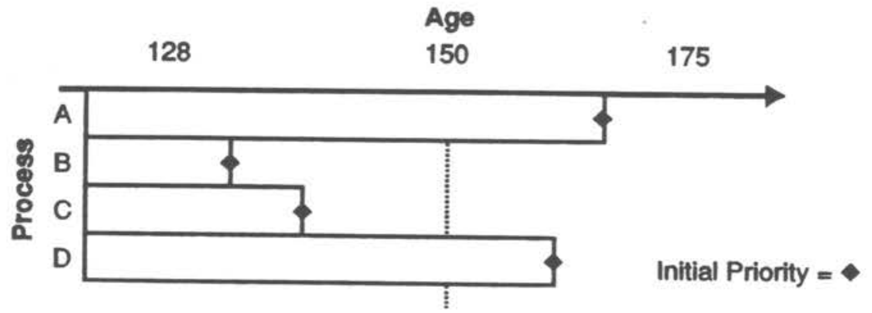


Figure 4a: Four processes are in the active queue. Processes A and D have priorities greater than 150, and processes B and C have priorities less than 150. If D_MaxAge is set at 150, processes B and C will continue to age until they reach 150. Processes A and D do not age. As a result, process D are not executed until process A sleeps, waits, or terminates. If both process A and process D are sleeping, waiting, or have terminated, processes B and C are eligible for execution.

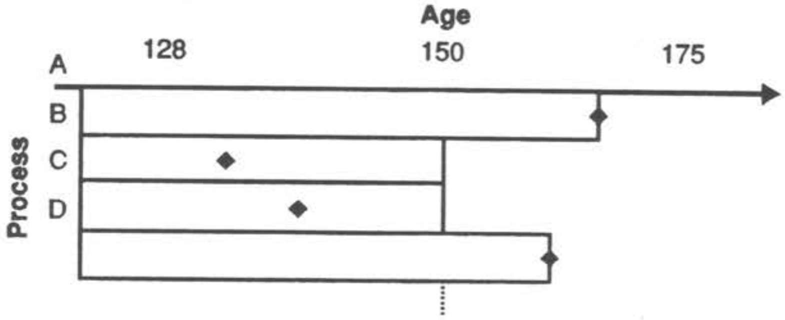


Figure 4b: After several time-slices have passed, processes B and C have both reached an age of 150, but they are not eligible for execution unless the other two processes are sleeping, waiting, or have terminated. They cannot age beyond 150.

NOTE: Any process performing a system call is not pre-empted until the call is finished, unless the process voluntarily gives up its time-slice. This exception is made because these processes may be executing critical routines that affect shared system resources and could be blocking other unrelated processes.

NOTE: System state processes are not time-sliced.

Example

For this example, four processes are run. One process has a priority of 130; the other three processes have a priority of 128. When the processes have been in the active queue for 9 minutes 1.60 seconds, a procs command is performed to determine the amount of time each process has spent in the CPU:

```
Id PId Grp.Usr  Prior  MemSiz Sig S  CPU Time   Age Module & I/O
 3   7   0.151   128    4.00k   0 w      1.48 24:09 shell <>>t11 >r0
 8   3   0.151   130    8.00k   0 a   3:00.52  0:09 a <>>>t11
 9   3   0.151   128    8.00k   0 a   2:00.44  0:09 b <>>>t11
10   3   0.151   128    8.00k   0 a   2:00.37  0:09 c <>>>t11
11   3   0.151   128    8.00k   0 a   2:00.27  0:09 d <>>>t11
12   3   0.151   128   20.00k   0 *      0.04  0:00 procs <>>t11 >r0
```

The following chart is a breakdown of how long each of the processes stayed in the CPU:

Process	Prior	Time in CPU	Percentage of CPU Time	
A	130	3 minutes 00.52 seconds	.3333	3/9
B	128	2 minutes 00.44 seconds	.2223	2/9
C	128	2 minutes 00.37 seconds	.2222	2/9
D	128	2 minutes 00.27 seconds	.2220	2/9

OS-9 Memory Management

If any object (a program, constant table, etc.) is to be loaded into memory, it must use the standard OS-9 memory module format. This enables OS-9 to maintain a module directory to keep track of modules in memory. The module directory contains the name, address, and other related information about each module in memory.

When a module is loaded into memory, it is added to the module directory. Each directory entry contains a ***link count***. The link count is the number of processes using the module.

When a process links to a module in memory, the module's link count is incremented by one. When a process unlinks from a module, the module's link count is decremented by one. When a module's link count becomes zero, its memory is de-allocated and it is removed from the module directory, unless the module is "sticky."

A ***sticky module*** is not removed from memory until its link count becomes -1 or memory is required for another use. A module is sticky if the sixth bit of the module header's M$Attr field is set. You can set this bit at link time or with the fixmod utility.

OS-9 Memory Map

OS-9 uses a software memory management system where all memory is contained within a single memory map. Therefore, all user tasks share a common address space.

A map of a typical OS-9 memory space is shown in Figure 2. The various sections shown for ROM, RAM, I/O, etc., are not required to be at specific addresses, unless noted otherwise. Microware recommends that you keep each section in contiguous reserved blocks arranged in an order that facilitates future expansion. It is always advantageous for RAM to be physically contiguous whenever possible.

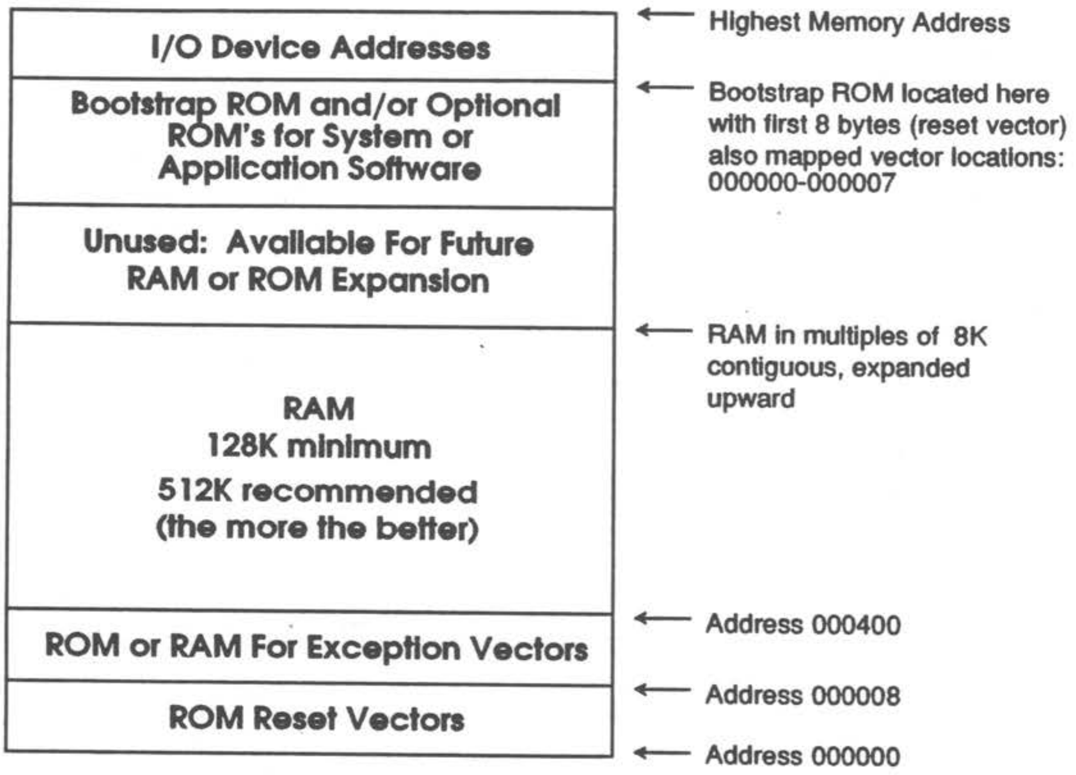

I/O Device Addresses
Bootstrap ROM and/or Optional ROM's for System or Application Software
Unused: Available For Future RAM or ROM Expansion
RAM
128K minimum
512K recommended
(the more the better)
ROM or RAM For Exception Vectors
ROM Reset Vectors
Highest Memory Address
Bootstrap ROM located here with first 8 bytes (reset vector) also mapped vector locations: 000000-000007
RAM in multiples of 8K contiguous, expanded upward
Address 000400
Address 000008
Address 000000

Figure 2: Typical OS-9 Memory Map

NOTE: For the 68020 and 68030 CPUs, the Vector Base Register (VBR) can be set anywhere in the system. Thus, for these types of systems, RAM or ROM are not required to be at address 0.

System Memory Allocation

During the OS-9 start-up sequence, blocks of RAM and ROM are found by an automatic search function in the kernel and the boot ROM. OS-9 reserves some RAM for its own data structures. ROM blocks are searched for valid OS-9 ROM modules.

The amount of memory required by OS-9 is variable. Actual requirements depend on the system configuration and the number of active tasks and open files.

User Memory

All unused RAM memory is assigned to a free memory pool. Memory space is removed and returned to the pool as it is allocated or de-allocated for various purposes. OS-9 automatically assigns memory from the free memory pool whenever any of the following occur:

- Modules are loaded into RAM.
- New processes are created.
- Processes request additional RAM.
- OS-9 requires more I/O buffers or its internal data structures must be expanded.

Storage for user program object code modules and data space is dynamically allocated from and de-allocated to the free memory pool. User object code modules are also automatically shared if two or more tasks execute the same object program. User object code application programs can also be stored in ROM memory.

The total memory required for user memory depends largely on the application software. It is suggested that a system minimum of 128K plus an additional 64K per user be available. Alternatively, a small ROM-based control system might only need 48K of memory.

Memory Fragmentation

Once a program is loaded, it must remain at the address where it was originally loaded. Although position-independent programs can be initially placed at any address where free memory is available, program modules cannot be relocated dynamically afterwards. This characteristic can lead to a sometimes troublesome phenomenon called ***memory fragmentation***.

When programs are loaded, they are assigned the first sufficiently large block of memory at the highest address possible in the address space. (If a "colored" memory request is made, this may not be true.) If a number of program modules are loaded, and subsequently one or more non-contiguous modules are "unlinked," several fragments of free memory space will exist. The total free memory space may be quite large. However, because the unused space is scattered, not enough space will exist in a single block to load a particular program module.

One way to avoid memory fragmentation is to load modules at system startup. This places the modules in contiguous memory space. Another way is to initialize each standard device when the system is booted. This allows the devices to allocate memory from higher RAM than would be available if the devices were initialized later.

If serious memory fragmentation does occur, the system administrator can kill processes and unlink modules in ascending order of importance until there is sufficient contiguous memory. You can determine the number and size of free memory blocks with the mfree utility

Colored Memory

OS-9 colored memory allows a system to recognize different memory types and reserve areas for special purposes. For example, you could design a part of a system's RAM to store video images and battery back up another part. The kernel allows isolation and specific access of areas of RAM like these. You can request specific memory types or "colors" when allocating memory buffers, creating modules in memory, or loading modules into memory. If a specific type of memory is not available, the kernel returns error #237, E$NoRAM.

Colored memory lists are not essential on systems with RAM consisting of one homogeneous type, although they can improve system performance on some systems and allow greater flexibility in configuring memory search areas. The default memory allocation requests are still appropriate for most homogeneous systems and for applications which do not require one memory type over another. Colored memory lists are required for F$Trans (address translation).

Colored Memory Definition List

The kernel must have a description of the CPU's address space to make use of the colored memory routines. You can establish colored memory by including a colored memory definition list (MemList) in the systype.d file, which then becomes part of the Init module. The list describes each memory region's characteristics. The kernel searches each region in the list for RAM during system startup.

A colored memory definition list contains the following information:

- Memory Color (type)
- Memory Priority
- Memory Access Permissions
- Local Bus Address
- Block Size the kernel's coldstart routine uses to search the area for RAM or ROM
- External Bus Translation Address (for DMA, dual-ported RAM, etc.)
- Optional name

The memory list may contain as many regions as needed. If no list is specified, the kernel automatically creates one region that describes the memory found by the bootstrap ROM.

MemList is a series of MemType macros defined in systype.d and used by init.a. Each line in the MemList must contain all the following parameters, in order:

type, priority, attributes, blksiz, addr begin, addr end, name, DMA-offset

```
MemList
  MemType  SYSRAM,255,B_USER,4096,0,$200000,OnBoard,$200000
  MemType  SYSRAM,250,B_USER+B_Parity,4096,$600000,$800000,OffBoard,0

[redacted]

OnBoard   dc.b "fast on-board RAM",0
OffBoard  dc.b "VME Bus Memory",0
```

_lcalloc() Allocate Storage for Array (low-overhead)

SYNOPSIS:

```
void *_lcalloc(nel,elsize)
unsigned long  nel,        /* number of elements in array */
               elsize;     /* size of elements */
```

FUNCTION: _lcalloc() allocates space for an array. nel is the number of elements in the array, and elsize is the size of each element. The allocated memory is cleared to zeros.

This function calls _lmalloc() to allocate memory. If the allocation is successful, _lcalloc() returns a pointer to the area. If the allocation fails, _lcalloc() returns zero (NULL).

NOTE: Use of the low-overhead allocation functions (_lcalloc(), _lmalloc(), _lrealloc()) instead of the general allocation functions (calloc(), malloc(), realloc()) saves eight bytes per allocation because the low-overhead functions do not save the allocation size or the four-byte check value.

CAVEATS: Use extreme care to ensure that only the memory assigned is accessed. Modifying addresses immediately above or below the assigned memory causes unpredictable program results.

The low-overhead functions require that the *programmer* keep track of the sizes of allocated spaces in memory. (Note the _lfree() and _lrealloc() parameters.)

_lfree() Return Memory (low-overhead)

SYNOPSIS:

```
void _lfree(ptr, size)
void           *ptr;   /* pointer to memory to be returned */
unsigned long  size;   /* size of memory to be returned */
```

FUNCTION: _lfree() returns a block of memory granted by _lcalloc() or _lmalloc(). The memory is returned to a pool of memory for later re-use by _lcalloc() or _lmalloc().

_lfree() never returns an error.

NOTE: Use of the low-overhead allocation functions (_lcalloc(), lmalloc(), _lrealloc()) instead of the general allocation functions (calloc(), malloc(), realloc()) saves eight bytes per allocation because the low-overhead functions do not save the allocation size or the four-byte check value.

CAVEATS: If _lfree() is used with something other than a pointer returned by _lmalloc() or _lcalloc(), the memory lists maintained by _lmalloc() are corrupted and programs may behave unpredictably.

The low-overhead functions require that the *programmer* keep track of the sizes of allocated spaces in memory.

_lmalloc() Allocate Memory from an Arena (low-overhead)

SYNOPSIS:

```
void *_lmalloc(size)
unsigned long   size;   /* size of memory block to allocate */
```

FUNCTION: _lmalloc() returns a pointer to a block of memory of size bytes. The pointer is suitably aligned for storage of data of any type.

[illegible]

NOTE: Use of the low-overhead allocation functions (_lcalloc(), _lmalloc(), _lrealloc()) instead of the general allocation functions (calloc(), malloc(), realloc()) saves eight bytes per allocation because the low-overhead functions do not save the allocation size or the four-byte check value.

CAVEATS: Use extreme care to ensure that only the memory assigned by _lmalloc() is accessed. Modifying addresses immediately above or below the assigned memory or passing _lfree() a value not assigned by _lmalloc() causes unpredictable program results.

The low-overhead functions require that the *programmer* keep track of the sizes of allocated spaces in memory. (Note the _lfree() and _lrealloc() parameters.)

ealloc() Resize a Block of Memory (low-overhead)

S :

```
void *_lrealloc(oldptr, newsize, oldsize)
void            *oldptr;   /* old pointer to block of memory */
unsigned long   newsize;   /* size of new memory block */
                oldsize;   /* size of old memory block */
```

ION: _lrealloc() re-sizes a block of memory pointed to by oldptr. oldptr should be a value returned by a previous _lmalloc(), _lcalloc() or _lrealloc().

_lrealloc() returns a pointer to a new block of memory. The size of this new block is specified by newsize. The pointer is aligned to store data of any type.

If newsize is smaller than oldsize, the contents of the old block are truncated and placed in the new block. Otherwise, the entirety of the old block's contents begin the new block.

The results of _lrealloc(NULL,newsize,0) and _lmalloc(size) are the same.

_lrealloc() returns zero (NULL) if the requested memory is not available or newsize is specified as zero.

NOTE: Use of the low-overhead allocation functions (_lcalloc(), _lmalloc(), _lrealloc()) instead of the general allocation functions (calloc(), malloc(), realloc()) saves eight bytes per allocation because the low-overhead functions do not save the allocation size or the four-byte check value.

_lmalloc() — Allocate Memory from an Arena (low-overhead)

SYNOPSIS:

```
void *_lmalloc(size)
unsigned long   size;   /* size of memory block to allocate */
```

FUNCTION: _lmalloc() returns a pointer to a block of memory of size bytes. The pointer is suitably aligned for storage of data of any type.

_lmalloc() maintains an amount of memory called an ***arena*** from which it grants memory requests. _lmalloc() will search its arena for a block of free memory large enough for the request and, in the process, coalesce adjacent blocks of free space returned by the _lfree() function. If sufficient memory is not available in the arena, _lmalloc() calls _srqmem() to get more memory from the system.

_lmalloc() returns zero (NULL) if there is no available memory or if the arena is detected to be corrupted.

NOTE: Use of the low-overhead allocation functions (_lcalloc(), _lmalloc(), _lrealloc()) instead of the general allocation functions (calloc(), malloc(), realloc()) saves eight bytes per allocation because the low-overhead functions do not save the allocation size or the four-byte check value.

CAVEATS: Use extreme care to ensure that only the memory assigned by _lmalloc() is accessed. Modifying addresses immediately above or below the assigned memory or passing _lfree() a value not assigned by _lmalloc() causes unpredictable program results.

The low-overhead functions require that the *programmer* keep track of the sizes of allocated spaces in memory. (Note the _lfree() and _lrealloc() parameters.)

_lrealloc() — Resize a Block of Memory (low-overhead)

SYNOPSIS :

```
void *_lrealloc(oldptr, newsize, oldsize)
void            *oldptr;    /* old pointer to block of memory */
unsigned long   newsize;    /* size of new memory block */
                oldsize;    /* size of old memory block */
```

FUNCTION: _lrealloc() re-sizes a block of memory pointed to by oldptr. oldptr should be a value returned by a previous _lmalloc(), _lcalloc() or _lrealloc().

_lrealloc() returns a pointer to a new block of memory. The size of this new block is specified by newsize. The pointer is aligned to store data of any type.

If newsize is smaller than oldsize, the contents of the old block are truncated and placed in the new block. Otherwise, the entirety of the old block's contents begin the new block.

The results of _lrealloc(NULL,newsize,0) and _lmalloc(size) are the same.

_lrealloc() returns zero (NULL) if the requested memory is not available or newsize is specified as zero.

NOTE: Use of the low-overhead allocation functions (_lcalloc(), _lmalloc(), _lrealloc()) instead of the general allocation functions (calloc(), malloc(), realloc()) saves eight bytes per allocation because the low-overhead functions do not save the allocation size or the four-byte check value.

CAVEAT: The low-overhead functions require that the *programmer* keep track of the sizes of allocated spaces in memory.

_mallocmin() Set Minimum Allocation Size

SYNOPSIS:

```
_mallocmin(size)
unsigned size;            /* minimum allocation size in bytes */
```

FUNCTION: _mallocmin() sets the minimum amount of memory that allocation functions may request through srqmem(). The size parameter cannot be less than the system memory block size. If a smaller size is requested, size is automatically set to the system memory block size.

OS-9 allows each process only 32 different memory segments; therefore, size should be increased if a program requires a great amount of memory. The extra space may be necessary if memory is fragmented.

_mallocmin() never returns an error.

_srqmem() System Memory Request

SYNOPSIS:

```
char *_srqmem(size)
unsigned size;            /* requested number of bytes */
```

FUNCTION: When tight control over memory allocation is required, _srqmem() and the complementary function _srtmem() are provided to request and return system memory.

This function is a direct hook into the OS-9 F$SRqMem system call. The specified size is rounded to a system-defined block size. A size of 0xffffffff obtains the largest contiguous block of free memory in the system. The global unsigned variable _srqsiz may be examined to determine the actual size of the block allocated.

If successful, a pointer to the memory granted is returned. If the request was not granted, _srqmem() returns the value -1 and the appropriate error code is left in the global variable errno.

The pointer returned always begins on an even byte boundary. Take care to preserve the value of the pointer if the memory is to be returned via _srtmem().

CAVEATS: The F$SrqMem request is intended for system level use, but the extended addressing range of the 68000 required some method to obtain memory without regard to where the memory is physically located.

A user process may have up to 32 non-contiguous F$SrqMem requests active at a given time. Ideally, the requests should be as large as practical, and preferably some multiple of 1K.

_srtmem() System Memory Return

SYNOPSIS:

```
int _srtmem(size,ptr)
unsigned size;            /* number of bytes to return */
char *ptr;                /* pointer to memory to return */
```

FUNCTION: _srtmem() is a direct hook into the OS-9 F$SRtMem system call. It is used to return memory granted by _srqmem(). Care should be taken to ensure that the size and ptr are the same as those returned by _srqmem().

If an error occurs, the function returns the value -1 and the appropriate error code is placed in the global variable errno.

calloc() Allocate Storage for Array

SYNOPSIS:

```
char *calloc(nel,elsize)
unsigned nel,           /* number of elements in array */
         elsize;        /* size of elements */
```

FUNCTION: calloc() allocates space for an array. nel is the number of elements in the array, and elsize is the size of each element. The allocated memory is cleared to zeroes. calloc() calls malloc() to allocate memory. If the allocation is successful, calloc() returns a pointer to the area. If the allocation fails, 0 is returned.

CAVEATS: Use extreme care to ensure that only the memory assigned is accessed. To modify addresses immediately above or below the assigned memory is sure to cause unpredictable program results.

ebrk() Obtain External Memory

SYNOPSIS:

```
extern int _memmins;
char *ebrk(size)
unsigned size;          /* number of bytes to return */
```

FUNCTION: ebrk() returns a specified amount of memory (size). The memory is obtained from the system via the F$SrqMem system request. It is intended for general purpose memory allocation.

The blocks of memory returned by this call may not be contiguous, thereby providing the ability to obtain a block of memory of a given size from anywhere in the 68000 address space.

To reduce the overhead involved in requesting small quantities of memory, ebrk() requests memory from the system in a minimum size determined by the global variable _memmins which is initially set to 8192, and satisfy the user requests from this memory space. ebrk() grants memory requests from this memory space provided the requests are no larger than the amount of space.

If the request is larger than the available space, ebrk() wastes the rest of the space and tries to get enough memory from the system to satisfy the request. This method works very well for programs that need to get large amounts of not necessarily contiguous memory in little bits and cannot afford the overhead of malloc(). Changing the _memmins variable causes ebrk() to use that value as the F$SrqMem memory request size.

If the memory request is granted, a pointer (even-byte aligned) to the block is returned. If the request is not granted, -1 is returned and the appropriate error code is placed in the global variable errno.

CAVEATS: The memory obtained from ebrk() is not given back until the process terminates.

free() Return Memory

SYNOPSIS:

```
free(ptr)
char *ptr;              /* pointer to memory to be returned */
```

FUNCTION: free() returns a block of memory granted by malloc() or calloc(). The memory is returned to a pool of memory for later re-use by malloc() or free(). The memory freed by malloc() or free() is returned to the system.

CAVEATS: It is dangerous to use free() with something other than a pointer previously returned by malloc() or calloc(). To do so hopelessly corrupts the memory lists maintained by malloc(), rendering them useless and possibly causing unpredictable program behavior.

freemem() — Determine Size of Unused Stack Area

SYNOPSIS:

```
int freemem()
```

FUNCTION: freemem() returns the number of bytes allocated for the stack that have not been used.

If compiler stack checking is enabled, the stack is checked for possible overflow before a function is entered. The lowest address the stack pointer has reached is retained so freemem() can report the number of bytes between the stack limit and the lowest stack value as the unused stack memory.

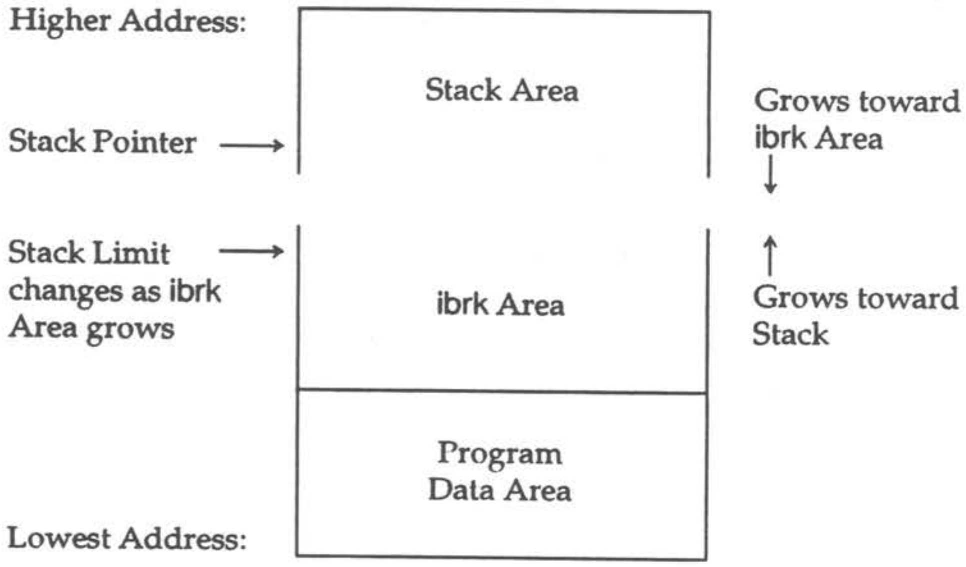
CAVEATS: The program must be compiled with stack checking code in effect for freemem() to return a correct result. This function is historical; avoid using it in new code as it is likely to be removed in a future release.

ibrk() — Request Internal Memory

SYNOPSIS:

```
char *ibrk(size)
unsigned size;                /* size of memory block */
```

FUNCTION: ibrk() returns a pointer to a block of memory of size bytes. The returned pointer is aligned to a word boundary. The memory from which ibrk() grants requests is the area between the end of the data allocation and the stack:



If the requested size would cause the ibrk area to cross the stack pointer, the request fails. You can use freemem() to determine the amount of stack remaining which is also the remaining ibrk area.

ibrk() is useful to obtain memory from a fixed amount of memory, unlike ebrk() whose available memory is that of the entire system. The C I/O library functions request the first 2K of I/O buffers from this area, the remainder from ebrk().

CAVEATS: Be very careful not to crowd out the stack with ibrk() calls. When stack checking is in effect, the program aborts with a ***Stack Overflow*** message if insufficient stack area exists to call a function.

malloc() — Allocate Memory from an Area

SYNOPSIS:

```
char *malloc(size)
unsigned size;          /* size of memory block to allocate */
```

FUNCTION: malloc() returns a pointer to a block of memory of size bytes. The pointer is suitably aligned for storing any type of data.

malloc() maintains an amount of memory called an ***arena*** from which it grants memory requests. malloc() searches its arena for a block of free memory large enough for the request and, in the process, unites adjacent blocks of free space returned by the free() function. If insufficient memory is available in the arena, malloc() calls ebrk() to get more memory from the system.

malloc() returns NULL (0) if there is no available memory or if it detects that the arena is corrupted by storing outside the bounds of an assigned block.

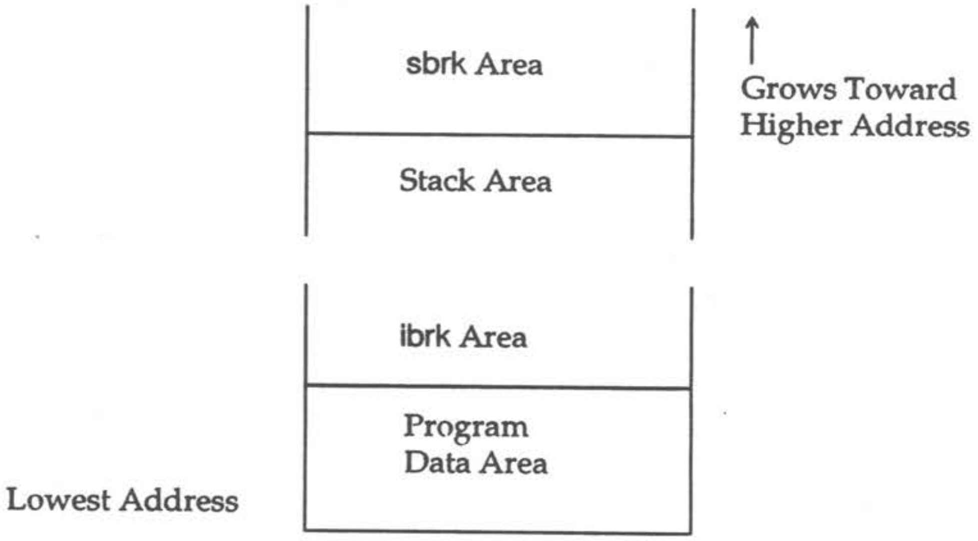
CAVEATS: Use extreme care to ensure that only the memory assigned by malloc() is accessed. Modifying addresses immediately above or below the assigned memory or passing free(), a value not assigned by malloc(), causes unpredictable program results.

sbrk() — Extend Data Memory Segment

SYNOPSIS:

```
char *sbrk(size)
unsigned size;          /* size of memory block desired */
```

FUNCTION: sbrk() allocates memory from the top of the data area upwards.



sbrk() grants memory requests by calling the F$Mem system call. This method resizes the data area to a larger size; the new memory granted is contiguous with the end of the previous data memory.

On systems without an MMU, this call is certain to fail quickly, because it may keep growing in size until the data area reaches other allocated memory. At this point, it is impossible to increase in size and an error is returned. A program may be able to increase its data size only 20K, even if there is 200K available elsewhere.

To gain the most utility of the 68000 addressing space, use the ebrk() function which returns pointers to memory no matter where it is located in the system.

srqcmem() — Allocate Colored Memory

SYNOPSIS:

```
#include <memory.h>
char *srqcmem(bytecnt, memtype)
int bytecnt,            /* size of memory to allocate */
    memtype;            /* type of memory to allocate */
```

FUNCTION: srqcmem() is a direct hook to the F$SRqCMem system call. bytecnt is rounded to a system-defined block size. The size of the allocated block is stored in the global integer variable _srqcsiz. If bytecnt is 0xffffffff, the largest contiguous block of free memory in the system is allocated.

memtype indicates the specific type of memory to allocate. <memory.h> contains definitions of the three types of memory that you may specify:

SYSRAM	System RAM memory
VIDEO1	Video memory for plane A
VIDEO2	Video memory for plane B

If memtype is zero, no memory type is specified. Consequently, any available system memory may be allocated.

If successful, a pointer to the memory granted is returned. The pointer returned always begins on an even byte boundary. If the request was not granted, the function returns the value -1 and the appropriate error code is placed in the global variable errno.

NOTE: srqcmem() is identical to _srqmem() with the exception of the additional color parameter.

stacksiz() — Obtain Size of Stack Used

SYNOPSIS:

```
int stacksiz()
```

FUNCTION: If the stack checking code is in effect, a call to stacksiz() returns the maximum number of bytes of stack used at the time of the call. You can use this function to determine the stack size a program requires.

NOTE: This function is historical and will likely be removed in a future release.

Interfacing to Assembly Language

C programs can run hand-written assembly language either by in-line coding in C programs using the #asm and #endasm directives or by giving the name of previously assembled relocatable file(s) on the C executive command line.

It is difficult to determine just where the compiler is during code generation. Therefore, you should not embed assembly language code within C functions. Assembly code is best placed in a separately assembled module to be linked with the rest of the program or in a stand-alone function without any C code.

If the c68 compiler is used, all functions and machine language subroutines are called by bsr instructions, except functions more than 32K away from the current pc.

The register usage conventions are (in general) as follows:

`D0 - D1`	Function argument/return registers
`D2 - D3`	Compiler allocated temporaries
`D4 - D7`	Used for user register variables
`A0 - A1`	Compiler allocated temporaries
`A2 - A4`	Used for user register variables
`A5`	Frame pointer
`A6`	Base address of variable storage area
`A7`	Stack pointer

The compiler uses a complex register allocation method to provide the smallest, fastest code for the majority of programs encountered. The 68000 has a large number of processor registers. Exactly half of these are available for use as register variables. The compiler uses the remaining processor registers for storing the intermediate results during the evaluation of expressions.

The a6 register is used as a pointer to the base of the global (static) variables. It is passed to a program when the program is forked and is never changed by C code.

The parameter type and the order specified in the parameter list indicates to the called function where the parameter is located. Parameters are either in a register or on the stack.

For this discussion, an integral parameter is of type int, a pointer, or a char or short converted to an int. A double parameter is of type double or a float converted to a double. A float is converted to a double before being passed on the stack.

The first integral parameter is passed in d0, and the second integral parameter (if any) is passed in d1. A single double parameter is passed in d0 and d1, the most significant half in d0, the least significant half in d1. Any remaining parameters are pushed on to the stack. If the first parameter is integral and the second is a double, the integral parameter is passed in d0 and the double is passed entirely on the stack.

If a function is to return a value, the integral (or float) value is returned in the d0.l register. A double value is returned in d0.l and d1.l.

Examples of C Parameter Passing Techniques

Assumption Assembler	C Code	Generated
init i, j, k; double a, b, c;	func(i);	move.l i,d0 bsr func
	func(i,j);	move.l j,d1 move.l i,d0 bsr func
	func(i,j,k);	move.l k,-(sp) move.l j,d1 move.l i,d0 bsr func
	func(a)	movem.l a,d0/d1 bsr func
	func(a,b)	move.l b+4,-(sp) move.l b+0,-(sp) movem.l a,d0/d1 bsr func
	func(a,i)	move.l i,-(sp) movem.l a,d0/d1 bsr func
	func(i,a)	move.l a+4,-(sp) move.l a+0,-(sp) move.l i,d0 bsr func

All functions (C or assembler) are required to restore any changed registers to the values they contained when the function was called. The only exceptions to this are function return register(s) and register(s) in which the function's argument(s) are passed.

All parameters passed on the stack are 4-byte long words. Types char and short are sign extended to long words. Types unsigned char and unsigned short are padded to the left with zeroes.

The 68000 bsr instruction, which is used for function calling, is limited to a ±32K address displacement. The bsr instruction has sufficient range for all but the very largest programs. In order to permit function calls outside this range while retaining position independence of the code, the linker automatically builds a jump table of long addresses of function entry points outside the range of bsr.

This table resides in the data area. It is accessed by the symbol _jmptbl defined by the linker. When coding assembly language routines, all external functions should be accessed by the word length displacement form of bsr so the linker can change the bsr to a jump if the displacement is too distant.

Consult the ***OS-9/68000 Assembler/Linker/Debugger Manual*** for more information on the jump table.

Introduction to Trap Handlers

OS-9 provides support for the software-generated exception called a ***trap***. A process can install a module, called a ***trap handler***, on any one of the 16 vectors available. Because modules in OS-9 are re-entrant, multiple processes can simultaneously share a single trap handler. All of these attributes combined can save memory, allow for modular programming, and make software more flexible.

For more information on the TRAP instruction, refer to the appropriate 68000 assembly language programming manual.

Static Storage

The first step in creating a trap handling system is to write the trap handler itself. When writing a trap handler, you can assume that only one process is using the trap handler at one time. The operating system maintains this illusion by allocating a different trap handler static storage for each process. That is, a trap handler may have its own set of variables, and this set of variables is different for each process that links to it.

An important consideration is the way that trap handlers are executed. When a process calls a trap handler, it is as if that process had branched to a subroutine in its own module. A trap handler executes in the same ***process context*** as the process that called it; it does not execute as another process. If the trap handler calls sleep, the process goes to sleep. Figure 1 shows how a process calls a trap handler and how the trap handler returns to the calling process.

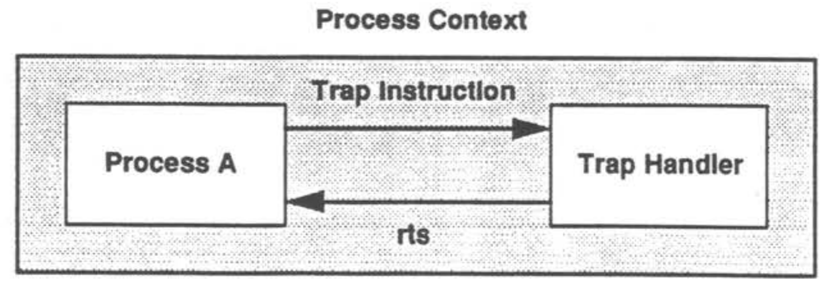


Figure 1: When process A executes a trap instruction, the trap handler is called. After executing its instructions, the trap handler returns to the process just as a subroutine would.

Trap Handler Construction

Trap handlers are constructed from three basic elements:

- Trap init code
- Trap handling code
- Trap terminate code

The trap handling code is usually broken into several units as well: code to dispatch the different trap function codes and the subroutines themselves. Figure 2 shows the parts of a trap handler and their normal organization.

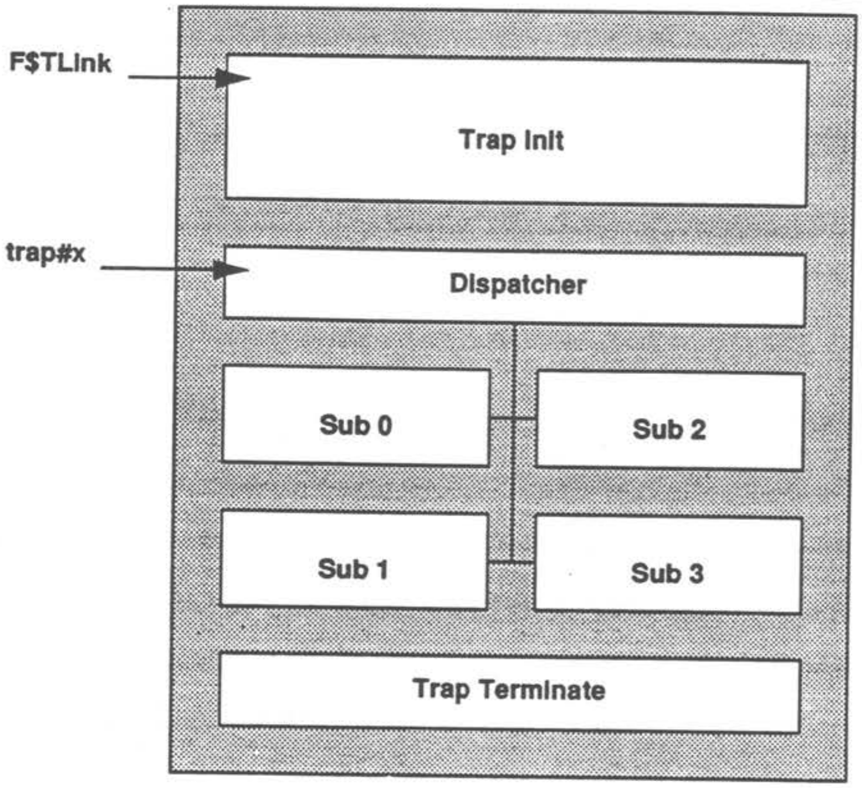


Figure 2: This is the form of a trap handler. Notice the two different entry points. Currently, Trap Terminate does not get executed. It is for future use only.

Trap Init Code

The trap init portion of a trap handler gets called when a process executes a F$TLink system call. This system call tells the operating system to install a specified trap handler on a specified vector. When F$TLink is executed, the kernel calls the ***trap init*** portion of the trap handler. This code should prepare the trap handler for future calls by doing such things as initializing constants or creating data tables. In other words, trap init should do these tasks so that the subroutines do not have to be concerned about them later.

Trap Handling Code

The actual trap handling code is normally constructed from two parts: the dispatcher and the subroutines themselves. The dispatcher is in charge of keying off the function code requested by the calling process and getting control to the proper subroutine. The dispatcher may also build a stack frame and set up registers for use by the subroutine. The subroutines themselves are trap handler specific.

There are two ways to return from a trap handler:

- Each subroutine can return to the calling process
- Each subroutine can return to the dispatcher that will, in turn, return to the calling process.

The second method is preferred because it allows the subroutine to be more modular. The most important concern related to returning from a trap handler, though, is getting the stack and registers restored. When returning from a trap handler, control goes directly to the user; it does not go through the kernel. Therefore, it is important to restore all registers and the stack pointer before returning to the calling process.

NOTE: While in a trap handler, register a6 points to the trap handler's static storage, not the calling process' static storage.

Trap Terminate Code

The calling of the trap terminate code has yet to be implemented. This code may eventually get called when a process finishes using a trap handler. For now, this code is never executed.

Sequence of Events

The following sequence of events occurs when a process uses a trap handler. Assume the process is test and the trap handler is mytrap.

① Test starts running.

② Test executes the F$TLink system call to install the trap handler:

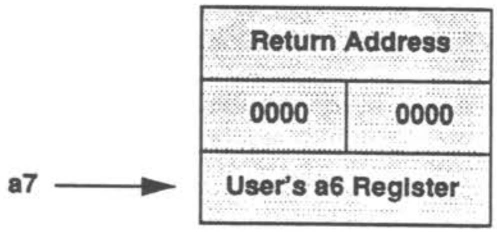
```
TrapName: dc.b "mytrap",0

    move.l #5,d0         set trap vector number
    move.l #0, d1        no extra memory for trap handler
    lea TrapName(a6), a0 move pointer to trap name into a0
    os9 F$TLink          install the trap handler
```

③ The kernel associates mytrap with vector #5 of test's trap vector table.

④ The kernel calls the trap init portion of the trap handler with the following register usage and stack frame:

d0.w: Vector on which the trap handler was installed.
d1.l: Additional static storage allocated for trap handler (optional).
d2-d7: User's registers at the time of the F$TLink system call.
a0: Trap handler module name pointer (updated).
a1: Trap handler execution entry point.
a2: Trap handler module pointer.
a3-a5: User's registers at the time of the call.
a6: Static storage base address.
a7: Stack pointer to the following stack frame:



⑤ Mytrap performs any necessary initialization, restores the caller's a6 register, and returns the following:

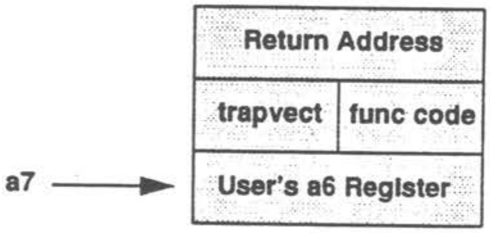
```
movea.l (sp),a6   restore user's a6 register
lea 8(sp),sp      position return address
rts               return from trap init portion of
                  the trap handler
```

⑥ Test executes for some time and calls the trap handler:

```
tcall 5,0          call the trap handler on vector 5
                   with function code 0
```

The kernel catches the trap exception and dispatches control to the trap handler's trap entry point with the following stack frame and register usage:

d0-d7:	User's registers at the time of the trap.
a0-a5:	User's registers at the time of the trap.
a6:	Pointer to the trap handler's static storage.
a7:	Pointer to the following stack frame:



⑦ Mytrap dispatches the call based on the function code passed in the stack frame:

```
cmpi.w #1,2(sp)     compare function code to 1
bgt.s TooHigh       if code is too large...
beq.s Sub1          if code == 1 ...
bra.s Sub0          if code == 0 ...
```

⑧ The subroutines perform their respective tasks and return to the caller:

```
Sub1:  add.l d0,d     add d0 to d1 with result in d1
                      (arbitrary task)
   move.l (sp),a6     restore user's a6 register
   addq #8,sp         position return address
   rts                return back to the user
```

⑨ When the process terminates, the associated trap handler static storage is deallocated.

Trap Vector Usage

When a process uses a trap handler, the process must install the trap handler on a vector. A process may use a total of 16 trap vectors. Three of these vectors are used by existing features of OS-9:

- Vector 0 is used by the kernel.
- Vector 13 is used by the math trap handler.
- Vector 15 is used by the cio trap handler.

The other 13 vectors are free for installation of user trap handlers.

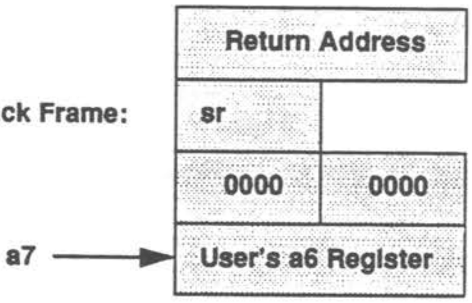
System-State Version 2.2 Trap Handlers

Trap handlers normally run in user state. Creating a system-state trap handler is the best way for a user-state process to perform some activities in system state. A system-state trap handler is constructed like a user-state trap handler, but it has a different module header and returning mechanism. In order to signal to the kernel that the trap handler needs to run in system state, the attributes/revision word in the module header must be defined as follows:

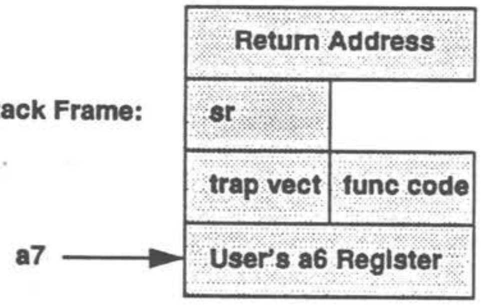
```
Attr_Rev:  equ  ((SupStat+ReEnt)<<8)+Rev  * re-ent is optional
```

The entry point stack frames are slightly different for a system-state trap handler as well. Each frame has an additional word just before the return address that contains an image of the status register as it should be restored before returning to the calling process. The following are the stack frames:

F$TLink Stack Frame:



Trap #x Stack Frame:



This difference in the stack frames changes the way the routines return from the trap handler. Instead of rts, rte should be used to restore both the pc and the sr. The programmer should also be aware of system-state programming concerns in general.

NOTE: For Version 2.3 of OS-9, the system-state trap handlers use the same diagrams as the user-state trap handlers.

The following is source code for an example trap handler and test program:

This is the trap handler:

```
       nam Trap Handler
       ttl Example trap handling module

       use defsfile

Type    equ (TrapLib<<8)+Objct
Revs    equ ReEnt<<8

       psect traphand,Type,Revs,0,0,TrapEnt

       dc.l TrapInit initialization entry point
       dc.l TrapTerm termination entry point
********************************************
* TrapInit: initialize the trap handler
*   Passed: d0.w - User trap number
*  d1.l -      (optional) additional static storage
*  d2-d7 -     caller's registers at the time of the trap
*  (a0) -  module name
*  (a1) -  trap handler execution entry point
*  (a2) -  trap module pointer
*  a3-a5 - caller's registers
*  (a6) -      pointer to static storage for trap handler
*  (a7) -  pointer to trap init stack frame
*
*
*
*
*
*
* Returns: (a0) -  updated trap handler name pointer
*       (a1) -         execution entry point
*       (a2) -         trap module pointer
*       cc -       carry set, d1.w - error code if error
TrapInit:  movem.l (a7),a6      restore a6
       addq.l #8,a7              take other stuff off the stack
       rts                return to caller
```

```
**********************************************
*  TrapEnt: trap handler entry point
*   Passed:     d0-d7 - caller's registers
*      a0-a5 - caller's registers
*      a6 -    pointer to the trap handler's static storage
*      a7 -    pointer to the following frame
*
*
*
*
*
*
* Returns: cc - carry set, d1.w - error code if error

          org 0
S.d0:     do.l 1 * this is a picture of the stack frame after the movem
S.d1:     do.l 1
S.a0:     do.l 1
S.a6:     do.l 1
S.func:   do.w 1
S.vect:   do.w 1
S.pc:     do.l 1

TrapEnt:  movem.l d0-d1/a0,-(a7)  save registers
          move.w S.func(a7),d0    get the function code
          cmp.w #1,d0             check it against 1
          bhi.s FuncErr           if it's higher then return error
          beq.s Trap10            if it's 1 go to Trap10
          lea String1(pc),a0      get address of string1
          bra.s Trap20            jump to writeln
Trap10:   lea String2(pc),a0      get address of string2
Trap20:   moveq #1,d0             write to standard out (path 1)
          moveq #80,d1            at most 80 characters
          os9 I$WritLn            call the writln
          bcs.s Abort             if error then reflect to user
Trap90:   movem.l (a7)+,d0-d1/a0/a6-a7 get our registers back
          rts                     return to user

FuncErr:  move.w #1<<8+99,d2      return bozo error
Abort:    move.w d1,S.d1+2(a7)    get the error into d1.w
          ori #Carry,ccr          set the carry
          bra.s Trap90            go to exit

String1:  dc.b "Microware Systems Corporation",C$CR,C$LF,0
String2:  dc.b "      Quality Keeps us #1",C$CR,C$LF,0
```

```
********************************************
* This code never gets called, but I'll return an error if the PC
* happens to get here by mistake.

TrapTerm:  move.w #1<<8+199,d1 return weird error
           os9 F$Exit          terminate caller

           ends
```

The following is the test program that uses the trap handler:

```
          nam TrapTest
          ttl Trap Testing Program

          use defsfile

Edition   equ 1
Typ_Lang  equ (Prgrm<<8)+Objct
Attr_Rev  equ (ReEnt<<8)+0

psect traptst_a,Typ_Lang,Attr_Rev,Edition,1024,Test

TrapNum:  equ 5
TrapName: dc.b "mytrap",0
*
* install the trap handler
*
Test:     moveq #TrapNum,d0       get the trap number
          move.l #0,d1            no  memory over-ride
          lea TrapName(pc),a0     get the name
          os9 F$TLink             link "mytrap" to this process
          bcs.s Test99            if error...
*
* call the trap handler twice
*
          tcall TrapNum,0         call trap with function code 0
          bcs.s Test99            if error...
          tcall TrapNum,1         call trap with function code 1
          bcs.s Test99            if error...
          moveq #0,d1             set for error-free return
Test99:   os9 F$Exit              terminate this process
ends
```

Extension Modules

To enhance OS-9's capabilties, additional modules can be executed at boot time. These ***extension modules*** provide a convenient way to install a new system call code or collection of system call codes (such as a system security module). The kernel calls the modules at boot time if their names are specified in the Extension list (the M$Extens offset) of the Init module and the kernel can locate them.

NOTE: Extension modules may only modify the d0, d1, and ccr registers.

To include an extension module in the system, you can either burn the module into ROM or complete the following steps:

① Assemble and link the module so that the final object code appears in the /h0/CMDS/BOOTOBJS directory.

② Create a new Init module:

- Change to the DEFS directory and edit the CONFIG macro in the systype.d file. The name of the new module must appear in the Init module extension list. For example, if the name of the new module is mine, the following line should be added immediately before the endm line:

```
Extens dc.b "os9p2 mine",0
```

  NOTE: Os9p2 is the name of the default extension module.

- Remake the Init module.

③ Create a new bootfile:

- Change to the /h0/CMDS/BOOTOBJS directory:

```
chd /h0/cmds/bootobjs
```

- Edit the bootlist file so that the extension module name appears in the list.
- Create a new bootfile with the os9gen utility. For example:

```
os9gen /h0fmt -z=bootlist
```

④ Reboot the system and check that the new module is operational.

The following code is an example of an OS9P2 module.

os9p2

```
            nam      os9p2
            ttl      extension module for os-9
****************************************************************
*     Extension to OS-9/68000
*     This code adds additional system calls to the kernel
**************************************************************
*     Edition History
*      #   date      Comments                       by
*     --- --------   --------------------------     ---
*      1  90/11/16   Created                        dwj
Edition     equ      1
            use      defsfile
Type        set      (Systm<<8)+Objct
Revs        set      ((ReEnt+SupStat)<<8)+0
            psect os9p2,Type,Revs,Edition,0,os9p2
**********************************************
*     os9p2 - initialization for os9p2
*     Passed:  (a3) = global storage used by calls
*              (a6) = system global pointer
*              (a7) = system stack pointer
*     Returns: (cc) = carry set if error
*              d1.w = error code if error
os9p2:   movem.l     d0/a1-a2,-(a7)
         lea         SrvcTbl(pc),a1
         os9         F$SSvc
         movem.l     (a7)+,d0/a1-a2
         rts
*************************************************
*     The service requests table
SrvcTbl:    dc.w        F$Coma,Coma-*-4
            dc.w        -1
*************************************************
*     Coma - put process to sleep indefinately
*     Passed by System:
*           (a3) = address of static data (unused)
*           (a4) = process descriptor
*           (a5) = address of caller's register stack
*           (a6) = system global variables
*     Passed by Caller:
*           none
*           Returns:
*           d0.l = value returned from F$Sleep
Coma:       move.l      d0,-(a7)
            moveq.l     #0,d0
            os9         F$Sleep
            move.l      d0,R$d0(a5)
            move.l      (a7)+,d0
            rts
            ends
```

makit

```
r68 os9p2.a -o=./RELS/os9p2.r
l68 ./RELS/os9p2.r-o=/h0/cmds/bootobjs/os9p2

r68 codes.a -o=./RELS/codes.r
attr -w /h0/lib/sys.l
rename /h0/lib/sys.l sys.l.org
merge /h0/lib/sys.l.org ./RELS/codes.r >/h0/lib/sys.l
```

codes.a

```
*****************************************************
*This file contains the user defined system calls

      psect codes,0,0,0,0,0

F$Coma:  equ   192

      ends
```

coma.c

```
#include <stdio.h>

int signal;
sighand(sig)
int sig;
{
      signal = sig;
}

main()
{
      int      sighand();

      intercept(sighand);
      printf("going to sleep\n");
      coma();
      printf("after the sleep\n");
}

#asm
**********************************
*            coma binding call

coma: link a5,#0
      os9  F$Coma
      unlk a5
      rts
#endasm
```

SCF Device Driver

The Sequential Character File Manager (SCF) is a re-entrant subroutine package for I/O service requests to devices which operate on a character-by-character basis, such as terminals, printers, and modems. SCF can handle any number of editing functions for line-oriented operations such as backspace, line delete, repeat line, auto line feed, screen pause, and return delay padding.

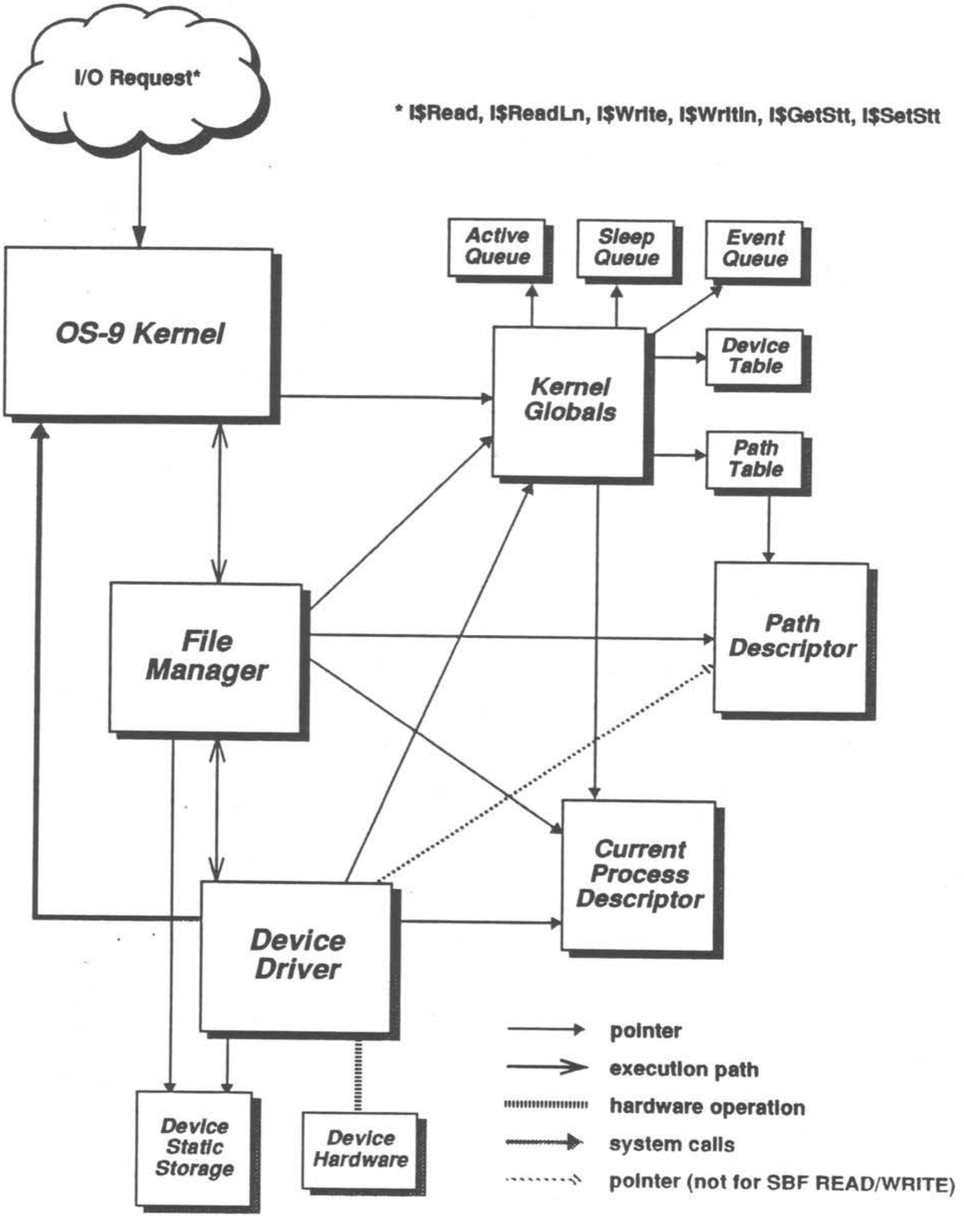
SCF device drivers support I/O devices that read and write data one character at a time, such as serial devices.

Generally, the input data (usually from a keyboard) is buffered by the driver's interrupt service routine. Each read request returns one character at a time from the driver's circular input FIFO buffer. If the buffer is empty when the request occurs, the driver must suspend the calling process until an input character is received. Input interrupts are usually enabled throughout the time the device is attached to the system. If the device is incapable of interrupt-driver operation, the driver must poll the device until the data becomes available. This situation has a harmful effect on real-time system performance.

The output data may or may not be buffered, depending on the physical characteristics of the output device. If the device is a memory-mapped video display driven by the main CPU, buffering and interrupts are not usually needed. If the device is a serial interface, you should use buffering and interrupts. Each write request passes a single output character to the driver. The character is placed in a circular FIFO output buffer. The output interrupt routine takes output characters from this buffer. If the buffer is full when a write request is made, the driver should suspend the calling process until the buffer empties sufficiently.

The I$GetStt system call (SS_Ready) and I$SetStt system call (SS_SSig) permit an application program to determine if the buffer contains any data. By checking first, the program is not suspended if data is not available.

The driver may optionally handle full input buffer conditions using X-ON/X-OFF or similar protocols. The input routine must also handle the special pause, abort, and quit control characters. All other control characters (such as backspace, line delete, etc.) are handled at the file manager level.



Typical system calls made by the driver include (if any):
F$Sleep, F$Event, F$CCtl, F$SRqMem, F$SRtMem

I/O System Layout for READ/WRITE/GETSTAT/SETSTAT Routines

INIT — Initialize Device and its Static Storage Area

INPUT:

```
(a1) = address of the device descriptor module
(a2) = address of device static storage
(a4) = process descriptor pointer
(a6) = system global data pointer
```

READ — Read Sector(s)

INPUT:

```
d0.1 = number of contiguous sectors to read
d2.1 = disk logical sector number to read
(a1) = address of path descriptor
(a2) = address of device static storage
(a4) = process descriptor pointer
(a5) = caller's register stack pointer
(a6) = system global data storage pointer
```

WRITE — Write Sector(s)

INPUT:

```
d0.1 = number of contiguous sectors to write
d2.1 = disk logical sector number
(a1) = address of the path descriptor
(a2) = address of the device static storage area
(a4) = process descriptor pointer
(a5) = caller's register stack pointer
(a6) = system global data storage pointer
```

GETSTAT/SETSTAT — Get/Set Device Status

INPUT:

```
d0.w = status code
(a1) = address of the path descriptor
(a2) = address of the device static storage area
(a4) = process descriptor pointer
(a5) = caller's register stack pointer
(a6) = system global data storage pointer
```

TERM — Terminate Device

INPUT:

```
(a1) = address of the device descriptor module
(a2) = address of device static storage area
(a6) = system global static storage pointer
```

The following is the pseudo-code representation for an SCF driver:

The Init Entry Point

Five tasks are performed at the Init entry point to the device driver:

① Initialize the static storage. Device drivers cannot have initialized static storage. Therefore, the code in the Init routine must manually initialize the static storage.

② Initialize the hardware from the device descriptor. The driver should read the information from the device descriptor and set up the device accordingly. This initializes items such as interrupt level, baud rate, parity, etc.

③ Disable output interrupts. Because most serial chips constantly generate output interrupts, interrupts are disabled until needed.

④ Install the device on the interrupt polling table via F$IRQ. This specifies the routine to be called when an interrupt occurs on a given sector.

⑤ Enable input IRQ's. This allows you to enter characters asynchronous to calls to the driver.

The Read Routine

The read routine consists of five steps.

① If X-OFF is received from the host and the input buffer is almost empty:

- Output X-ON character.
- Clear input halted flag.

This code deals with the X-ON/X-OFF protocol used by OS-9 drivers. If the input was stopped for any reason, this code checks to see if it can be resumed at full speed.

② If a SS_SSig is pending, return error E$NotRdy. If a process requests a signal to be sent on data ready, no other process may read from that port. This prevents the process with the signal pending from missing data.

③ If the data is ready, remove a character from the input buffer. If the data is not ready:

- Call the sleep routine to wait for an interrupt.
- Go to the read routine to process the character.

SCF drivers use both an input and an output buffer. The input buffer is filled by the input IRQ routine and emptied by the driver's read routine. The output buffer is filled by the driver's write routine and emptied by the output interrupt routine.

④ If an error occurs during an interrupt, return error E$Read. If an error, such as framing or parity, occurs that can only be caught by the interrupt service routines, the driver must check for the error at the next read posted to the driver.

⑤ Return without error to the file manager.

The Sleep Routine

The read and write routines can both call the sleep routine. The file manager cannot call this routine. This routine consists of three steps:

① Tell the IRQ routine to send a signal when an interrupt occurs.

② Sleep until a signal is received: sleep(0).

③ If a fatal signal is received or the process is condemned, the routine should return to the file manager. Otherwise, the routine should return to the caller.

The driver sleeps until an interrupt occurs which sends a signal to the driver and wakes it up. If a fatal signal is received or the process is condemned in the meantime, this routine passes control back to the file manager.

The Write Routine

The write routine consists of four steps:

① If there is no more room for data in the output buffer:

- Call the sleep routine to wait for room.
- Go to the write routine to finish the write request.

If the output IRQ routine cannot keep up with the driver, the driver must sleep and wait for it.

② Place the character in the output buffer. The output interrupt routine removes the character from the buffer and places it in the chip.

③ If the output is not halted and the buffer was empty, enable output interrupts.

④ Return to file manager.

The GetStat Routine

The GetStat routine consists of a single switch procedure: Switch(code). Code may be one of the following:

- Case SS_Ready: code returns the number of bytes in the input buffer.
- Case SS_EOF: procedure returns without error.
- By default, an Unknown Service Error (E$UnkSvc) is returned.

The SetStat Routine

The SetStat routine consists of a single switch procedure: Switch(code). Code may be one of the following:

- Case SS_SSig:
  - If SS_SSig is already pending, return E$NotRdy.
  - If the data is already available, send the requested signal.
  - If the data is not available, save the information. This allows the interrupt routine to send the signal.
  - Return to the file manager.
- Case SS_Relea:
  - Disable sending of signals.
  - Return to the file manager.
- By default, error E$UnkSvc is returned.

The Terminate Routine

The terminate routine consists of three steps:

① Wait for any pending output. If the driver places something in the output device and the process closes the device, the output is lost if the driver terminates before the output is complete.

② Disable all interrupts from the chip.

③ Remove the device from the interrupt polling table via F$IRQ.

Interrupt

The interrupt routine consists of two steps:

① Ensure that the interrupt belongs to the device. Because OS-9 allows multiple interrupt service routines on one vector, each IRQ routine must determine if the device it's responsible for generated the interrupt. If the interrupt does not belong to the device, return carry set.

② Switch (interrupt type)
- Input:
  - Switch (input character); input character may be:
    - Case NULL = break : no further checks.
    - Case V_INTR = send S$Intrpt to V_LPROC.
    - Case V_QUIT = send S$Abort to V_LPROC.
    - Case V_PCHR = set V_PAUS in statics.
    - Case V_XON = mark output resumed.
      - If there is something to output, enable output interrupts.
      - Return from interrupt.
    - Case V_XOFF = disable output interrupts
      - Mark output halted
      - Return from interrupt
    - By default, place the input char in the input buffer.
  - End switch.
  - If a buffer overrun occurs, mark the error in the device driver statistics.
  - If SS_SSig pending:
    - Send signal to the process.
    - Return from interrupt.
  - If the buffer is almost full, cause device to receive X-OFF.
  - Go to the WakeUp routine.
- Output:
  - If X-OFF or X-ON is to be sent:
    - Send character and mark it as sent.
    - Return from interrupt.
  - Remove the character from the driver's buffer.
  - Place the character in the hardware's output register.
  - If the output buffer is empty, disable the output interrupts.
  - Go to the WakeUp routine.

End switch.

The WakeUp Routine

The WakeUp routine consists of two steps:

① If the driver is sleeping, send an S$Wake signal to the driver.

② Return from interrupt.

RBF-Style Driver

The Random Block File Manager (RBF) is a re-entrant subroutine package for I/O service requests to random-access devices. RBF can handle any number or type of such devices simultaneously (such as large hard disk systems, small floppy systems, RAM disk systems, etc.) and is responsible for maintaining the logical file structure.

RBF reads and writes in 256-byte logical sectors. The file manager handles all file system processing and passes the driver a starting logical sector number (LSN), a sector count, and the address of the data buffer for each read or write operation.

The physical sector size of the media is assumed to be 256 bytes. If other physical sizes are used, it is the device driver's responsibility to translate and deblock the RBF LSNs into the media's LSNs. For example, if PD_SSize is set to 512 and a read request of 8 sectors at LSN 4 is made, the driver should translate the operation to a read of 4 sectors at LSN 2.

Read and write calls to the driver initiate the sector read/write operations and, if required, a prior seek operation.

If the controller cannot be interrupt-driven, it must wait until the media is ready, and then transfer the data by polling. If possible, you should avoid disk controllers that cannot be interrupt-driven. They cause the driver to dominate the system CPU while disk I/O is in progress.

For interrupt-driven systems, the driver initiates the I/O operations and suspends itself (F$Sleep or F$Event) until the interrupt arrives. The interrupt service routine then services the interrupt and "wakes up" the driver.

NOTE: If the driver is awakened by a signal (a keyboard abort for example) while waiting for the I/O interrupt to occur, it should suspend itself again until the I/O interrupt has occurred. This is because many read/write calls to a driver are made by RBF on behalf of itself (for example, in directory searching or bitmap updating). When a signal causes a process to terminate, RBF determines the appropriate time to return to the kernel. Failure to enforce the I/O interrupt completion may result in "locked" disks or corrupted media.

If DMA (Direct Memory Access) hardware support is available, I/O performance increases dramatically because the driver does not have to move the data between memory and the controller.

When the driver reads sector zero, it should copy the first 21 bytes of the sector into the drive table (PD_DTB) associated with the logical unit. Sector zero of the disk media has format information recorded by the format utility. This information allows the driver to determine the actual format of the media and to compare the device physical capabilities specified in the path descriptor options with the media format. This allows the driver to adapt its operation for reading and writing multiple formats in one physical drive. For example, a floppy drive that can read/write double-sided, double-density disks can be made to operate with single-sided or single-density media.

RBF always reads sector zero of the media when a file is opened. Many RBF drivers provide caching of sector zero to improve the performance of I$Open calls by RBF. This function is generally associated with media that is non-removable (such as hard disks). When a hard disk driver reads sector zero, it updates the drive table and copies the full sector zero into a local buffer. The state of the buffered sector for the unit is recorded in the logical unit drive table variables V_ZeroRd and V_ScZero. This enables the driver to return sector zero data on subsequent calls by RBF without accessing the disk. Removable media should not have sector zero buffered unless the driver is capable of automatically detecting the media removal (for example, by an interrupt).

GetStat calls to RBF devices are generally not seen by the driver. The majority of RBF GetStat calls are handled by RBF itself. Most RBF drivers ignore all GetStat calls.

SetStat calls to the driver are generally made for formatting operations (for example, restore head, write track) and sequencing down the disk (for example, head parking). Most RBF drivers ignore all other SetStat calls.

The INIT and TERM routines of RBF drivers are called directly by the kernel when the device is attached and detached. Typically, the INIT routine only performs controller-specific initialization such as adding the controller to the IRQ polling table, setting default values in the drive tables, and initializing the controller hardware interface.

NOTE: The INIT routine generally does not perform initialization of the logical units attached to the controller (such as the disk parameter definitions for SCSI drives). This type of initialization should normally be performed when the first Read/Write/GetStat/SetStat call is made to the unit.

The TERM routine typically disables the device's interrupts, if required, and removes the controller from the IRQ polling table.

The following is pseudo code for an RBF style driver:

The Init Routine

The Init routine consists of four steps:

① Initialize the static storage. You must manually initialize the static storage because device drivers cannot have initialized static storage.

② Initialize the drive tables with "fake values." This initializes the controller enough to get to sector zero to read the real information from the media.

③ Reset the hardware. Get the hardware in some known state to allow subsequent calls to the Read and Write routines to work properly.

④ Install the interrupt service routine on the interrupt polling table via F$IRQ.

The Read Routine

The read routine consists of five steps:

① If the controller is not initialized, initialize it.

② If not reading sector zero:

- Read the sectors into PD_BUF from the path descriptor.
- Return to the file manager with any errors that occurred.

The PD_BUF field of an RBF path descriptor points to an area of memory that the driver should use as a buffer.

③ If sector zero is not already buffered:

- Read sector zero into the sector zero buffer.
- Update drive table to reflect information from the media.

Most drivers buffer sector zero because RBF makes multiple requests to it.

④ Move the sector zero buffer to PD_BUF from the path descriptor.

⑤ Return to the file manager.

The Write Routine

The Write routine consists of three steps:

① If writing to sector zero and the format disable bit is set, return with a write error (E$Write). A process is not allowed to write to sector zero unless the disk is allowed to be formatted.

② Write the sectors to the disk.

③ If sector zero was written, mark the sector zero drive table as invalid. The next read of sector zero validates the buffer and the drive table.

The GetStat Routine

The GetStat routine returns an unknown service request error (E$UnkSvc). There are no valid GetStats in an RBF driver.

The SetStat Routine

The SetStat routine is switch(code). Code may be any of the following cases:

- Case SS_WTrk :
  - If format is disabled, return the format error E$Format.
  - If the controller is not initialized, initialize it.
  - Write track buffer to disk.
  - If a seek error occurs, return a seek error (E$Seek).
  - Return without error. If a non-seek error occurs, the driver still exits without error. The verification stage of the format should catch the other types of errors.
- Case SS_Reset:
  - If the controller is not initialized, initialize it.
  - Restore the heads to sector zero.
- The Unknown Service Error (E$UnkSvc) is returned by default.

The Terminate Routine

The terminate routine consists of two steps:

① Reset the controller to disable future interrupts.

② Remove the device from the interrupt polling table via F$IRQ.

The Interrupt Routine

The interrupt routine consists of two steps:

① Signal interrupt's occurrence to driver by clearing V_WAKE. When the driver wakes up, the driver verifies that the interrupt service routine woke it up.

② If the driver is sleeping (waiting for an interrupt), send a wake up signal to the driver (S$Wake). The driver should be sleeping every time an interrupt occurs. If the driver is not sleeping, a signal should not be sent.

ROM-Based OS-9/68000

The following is an explanation of how you can create a ROM based OS-9 system. You must have access to the equivalent of a Port-Pak because you need to make a new boot EPROM, a new Sysgo, and a new Init module. The location of the modules must be known by the boot EPROM because the modules which make OS-9 are not loaded from a boot disk. The Memory Search List, located in the **DEFS/systype.d** file, supplies the boot EPROM with this information.

The following shows an example of a search list for a ROM-based system:

```
Mem.Beg  equ  $02000000                              Beginning of memory
Mem.End  equ  $02100000                              Normal end of memory
Spc.Beg  equ  $00000000                              Start of ROM search area
Spc.End  equ  $00020000                              End of ROM search area
    .
    .
    .
**************************************************
*  Memory search definitions

MemDefs macro
    dc.l  Mem.Beg,Mem.End                            Normal RAM boundaries
    dc.l  0                                     Required to end RAM definitions
    dc.l  Spc.Beg,Spc.End               ROM boundaries (search here for modules)
    dc.l  0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,               Free bytes for patching
edm
```

This example shows a system with normal system RAM from $02000000 to $02100000 and ROM from $00000000 to $00020000. The ROM area can also contain the boot EPROM code (for example, ROM debugger, SysInit, etc.) as well as the OS-9 system modules. The ROM search routine skips over non-OS-9 modules (such as the boot EPROM code) and locates any valid modules in the ROM.

You can also have discontinuous RAM:

```
Mem.Beg   equ   $02000000                        Normal beginning of memory
Mem.End   equ   $02100000                           Normal end of memory
Mem.Beg1  equ   $03000000                  Beginning of discontinuous memory
Mem.End1  equ   $03100000                        End of discontinuous memory
Spc.Beg   equ   $00000000                           Start of ROM search area
Spc.End   equ   $00020000                             End of ROM search area
    .
    .
    .
*****************************************************
*  Memory search definitions

MemDefs macro
    dc.l  Mem.Beg,Mem.End,Mem.Beg1,Mem.End1       Normal RAM boundaries
    dc.l  0                                 Required to end RAM definitions
    dc.l  Spc.Beg,Spc.End           ROM boundaries (search here for modules)
    dc.l  0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,              Free bytes for patching
edm
```

This example shows a system with system RAM from $02000000 to $02100000 and from $03000000 to $03100000 and ROM from $00000000 to $00020000.

NOTE: These definitions must exactly match the target system.

You should also make changes to the systype.d file to indicate to the OS-9 kernel that there is no startup disk drive. Do this by changing the SysDev line in the CONFIG macro to be null. This is shown here:

```
*****************************************************
*  Configuration module constants
*  used only by the Init module

CONFIG macro
MainFram  dc.b  "Heurikon HK68/V20",0
SysStart  dc.b  "sysgo",0        Name of initial module to execute (version 2.4 )
                                                                    -or-
SysStart  dc.b  "sysgo_nodisk",0                                (version 2.3)

SysParam  dc.b  "",C$CR,0                      Parameters to initial process
SysDev set 0                                         No initial system disk
ConsolNm  dc.b  "/Term",0                          Console terminal pathlist
ClockNm   dc.b  "mk68901",0                                Clock module name
Extens    dc.b  "os9p2",0                                  Extension modules
endm
```

NOTE: For version 2.2 of the operating system, the SysDev line should be:

```
SysDev   dc.b "",0                                   No initial system disk
```

Also note that the startup module name is changed from the normal sysgo to sysgo_nodisk. The source code for sysgo_nodisk is included with the Port Pak and is located in the SYSMODS directory. With these changes made to the systype.d file, the modules which you must modify to make a ROM based system can be remade. You should execute the following commands:

① Change your current data directory to /h0/ROM:

```
chd /h0/rom
```

② Merge everything into one file with the make utility:

```
make debug -u
```

The object code will be located in ROM/RELS/debug.

③ Change your current data directory to /h0/SYSMODS:

```
chd /h0/sysmods
```

④ Use the make utility:

```
make sysgo_nodisk init -u
```

The object code is located in CMDS/BOOTOBJS/sysgo_nodisk and CMDS/BOOTOBJS/init.

You should also make a new boot EPROM using the object code file ROM/RELS/debug. Burn the necessary OS-9 modules into the EPROM. An example list of modules might be:

```
debug
Kernel
init
mk68901                                          the clock module
scf
pipeman
null
nil
sc68901                                          the serial driver
pipe
term
sysgo
shell
cio
any desired applications
```

This system could only run the shell. Any other utilities that are to be used must also be in the ROM. Remember that standard OS-9 utilities require the cio module.

Porting OS-9 to a New 68000 System

Before discussing how OS-9 is ported to a new system, you should understand two terms: the host system and the target system.

The ***host system*** is the development system used to edit and re-assemble OS-9. The host system can be any one of the following systems:

- A 68000 family-based computer with 512K RAM and OS-9/68000
- A VAX computer running Unix BSD4.2, Unix 4.3, or VMS 4.6
- An Apollo computer running Domain 10.x
- A Sun computer running Sun OS 3.x
- An HP9000 computer running HP-UX 6.3
- A Motorola Delta Box computer running MV68 Unix System V

The ***target system*** is the system on which you intend to install OS-9. The target system should consist of the following hardware:

- A 68000 family CPU
- At least 512K RAM
- At least 32K ROM capacity. Alternatively, you can use an emulator with 32K of overlay memory.
- Two serial I/O ports; one for a terminal and one for communications with the host system.

The 32K ROM is for convenience in bringing up OS-9. If the system is disk-based, the eventual target system can use as little as 8K for a boot ROM. The same is true of the two serial ports; they are not required after the porting process.

The target system may also be equipped with any other I/O devices which must eventually be supported by OS-9, although they will not be used in the initial installation steps.

You should hook up the serial ports that link the host to the target system. If you have some way to do it (for example, if you have existing software that already runs on your target system), test the communications link at this time.

The following figure shows the typical interconnection between the host and the target systems.

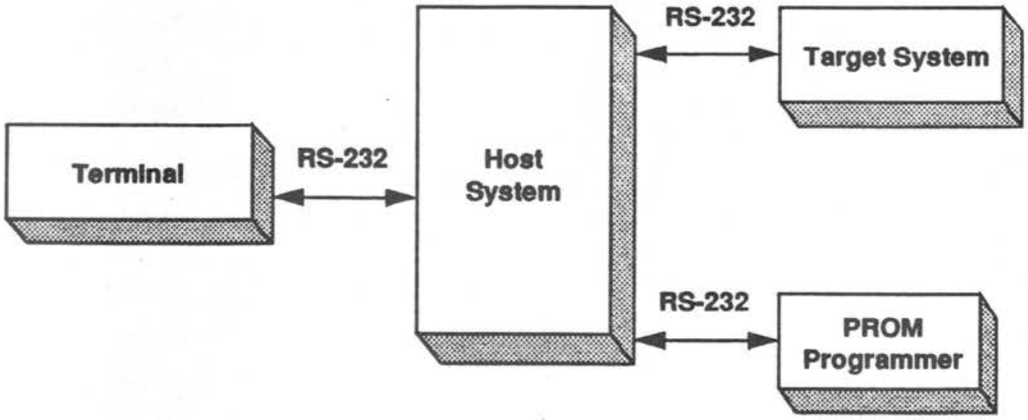


Figure 1: Typical Host and Target Interconnection
A programmer can use a terminal and work through a host system to set up and debug a target system. Once the target system has been debugged, terminals may be attached to it and it may be used like any other OS-9 system.

The following steps provide you with an overview of the porting operation:

① ***Preparation***
Before porting OS-9 to a new 68000 system, you should thoroughly know your hardware, software, and documentation. The more familiar you are with these, the easier your task will be. The following are some tips to follow before porting OS-9 to a new system:

- Prepare the target system hardware. Before attempting to install OS-9, make sure the hardware boots. If the target system is an untested prototype, use the assembler to make a simple, stand-alone test ROM that just prints a message on a terminal to verify basic hardware functionality.
- Check the distribution package. Familiarize yourself with the contents of the distribution package provided by Microware. Also verify that the distribution package is complete and that it is the correct version for the type of host system you are using.

② Modify the Systype.d File

Target system, hardware-dependent definitions should be concentrated in the systype.d file. This includes basic memory map information, exception vector methods (for example, vectors in RAM or ROM), I/O device controller memory addresses and initialization data, etc. Review and edit the systype.d file before you attempt to re-assemble any other routines.

Systype.d is included in the assembly of many other source files by the assembler's use directive. Your first editing job is to make a new systype.d file that describes your target system as closely as possible using the sample file provided in the distribution package. Some definitions are not used until later in the installation process, but it is recommended to set up everything in advance. Similarly, some of the definitions may not apply to your target system.

③ Write/Obtain a Console I/O Driver

You must create an OS-9 console I/O driver module for the console device. Microware may have an existing driver based on the same device your target system uses. If this is the case, little or no adaptation is required.

Otherwise, you must create a new console I/O driver module. It is easiest to modify an existing Microware-supplied serial driver. Refer to the ***OS-9 Technical Manual*** and the sample source files supplied for guidance.

A *device descriptor* module for the serial port is also needed. The console device driver module name is Term by convention. If the system has other identical serial I/O devices, they can be used if you make additional device descriptor modules for them. You should edit the standard makefile to suit the individual configuration needed to make these files.

④ Modify the Sysinit.a File

The sysinit.a file performs any special hardware initialization that your system may require after a reset or system re-boot occurs. It also determines whether the ROM debugger is enabled or not.

⑤ *Test Boot the System to the RomBug: Prompt*

To test boot the system, use the make utility to automatically assemble and link the component files to create a boot/debug ROM binary object file. Type the following command from the OSK/ROM file:

```
$ make debug
```

This command creates a ROM image that has a dummy boot and debugger with talk-through and download capabilities. The created binary output file is called debug.

If problems occur when you run make, check the following:

- Verify that systype.d is configured correctly.
- Verify that the "makefile" has the correct names of your customized files.

After the files are assembled and linked properly, make a ROM or load the code into the emulator overlay in memory.

NOTE: The linker output is a pure binary file. If your PROM programmer or emulator requires S-records, use the binex utility to convert the data.

If your PROM programmer cannot burn more than one PROM at a time and your system has the ROMs addressed as 16-bit or 32-bit wide memory, use the romsplit utility to convert the ROM object image into "8 bit wide" files.

⑥ *Write/Obtain a Serial Driver*

You must create an OS-9 serial driver module. Microware may have an existing driver based on the same device your target system uses. If this is the case, little or no adaptation is required.

Otherwise, you must create a new serial driver module. It is easiest to modify an existing Microware-supplied serial driver. Refer to the ***OS-9 Technical Manual*** and the sample source files supplied for guidance.

⑦ *Test Boot Up to the Shell Prompt*

⑧ *Write a RBF Driver, a Clock Driver, and a Disk Boot Driver*

You must create an OS-9 RBF driver module, a clock driver module, and a disk boot driver module. Microware may have existing drivers based on the same devices your target system uses. If this is the case, little or no adaptation is required.

Otherwise, you must create the new modules. It is easiest to modify the existing Microware-supplied serial drivers. Refer to the ***OS-9 Technical Manual*** and the sample source files supplied for guidance.

⑨ *Test the New System*

The quickest basic test of a new installation is to start using the system immediately. This usually reveals major problems if they exist.

⑩ *Take a Break*

Your new system should be up and running. You should now be ready for a break!