The Kernel

Responsibilities of the Kernel

The ***kernel*** is the nucleus of OS-9. It manages resources, controls processing, and supervises Input/Output. It is a ROMable, compact OS-9 module.

The kernel's primary responsibility is to process and coordinate system calls, or service requests. OS-9 has two general types of system calls:

- Calls that perform Input/Output, such as reads and writes.
- Calls that perform system functions. System functions include memory management, system initialization, process creation and scheduling, and exception/interrupt processing.

When a system call is made, a user trap to the kernel occurs. The kernel determines what type of system call the user wants to perform. It directly executes the calls that perform system functions, but does not execute I/O calls. The kernel provides the first level of processing for each I/O call, then completes the function as required by calling the appropriate file manager or driver.

For information on specific system calls, refer to the ***OS-9 System Calls*** section of this manual.

For specific information about creating new file managers, and examples which you can adapt to your specific system needs, refer to the ***OS-9 Technical I/O Manual***.

System Call Overview

For information about specific system calls, refer to ***OS-9 System Calls***.

User-state and System-state

To understand OS-9's system calls, you should be familiar with the two distinct OS-9 environments in which object code can be executed:

User-state The normal program environment in which processes execute. Generally, user-state processes do not deal directly with the specific hardware configuration of the system.

System-state The environment in which OS-9 system calls and interrupt service routines execute. On 68000-family processors, this is synonymous with supervisor state. System-state routines often deal with physical hardware present on a system.

Functions executing in system state have distinct advantages over those running in user state, including the following:

- A system-state routine has access to all of the processor's capabilities. For example, on memory protected systems, a system-state routine may access any memory location in the system. It may mask interrupts, alter OS-9 internal data structures, or take direct control of hardware interrupt vectors.
- Some OS-9 system calls are only accessible from system state.
- System-state routines are never time-sliced. Once a process enters system state, no other process executes until the system-state process finishes or goes to sleep (F$Sleep waiting for I/O). The only processing that may preempt a system-state routine is interrupt servicing.

System-state characteristics make it the only way to provide certain types of programming functions. For example, it is almost impossible to provide direct I/O to a physical device from user state. Not all programs, however, should run in system state. Reasons to use user-state processing rather than system-state processing include:

- User-state routines are time-sliced. In a multi-user environment, it is important to ensure that each user receives a fair share of the CPU time.
- Memory protection prevents user-state routines from accidentally damaging data structures they do not own.
- A user-state process can be aborted. If a system-state routine loses control, the entire system usually crashes.

- System-state routines are far more difficult and dangerous to debug than user-state routines. You can use the user-state debugger to find most user-state problems. Generally, system-state problems are much more difficult to find.
- User-state programs are essentially isolated from physical hardware. Because they are not concerned with I/O details, they are easier to write and port.

Installing System-state Routines

System-state routines have direct access to all system hardware, and have the power to take over the entire machine, crashing or hanging up the system. To help prevent this, OS-9 limits the methods of creating routines that operate in system state.

There are four ways to provide system-state routines:

- Install an OS9P2 module in the system bootstrap file or in ROM. During cold start, the OS-9 kernel links to this module, and if found, calls its execution entry point. The most likely thing for such a module to do is install new system call codes. The drawback to this method is that the OS9P2 module must be in ROM or in the bootfile when the system is bootstrapped.
- Use the I/O system as an entry into system state. File managers and device drivers always execute in system state. The most obvious reason to write system-state routines is to provide support for new hardware devices. It is possible to write a dummy device driver and use the I$GetStt or I$SetStt routines to provide a gateway to the driver.
- Write a trap handler module that executes in system state. For routines of limited use that are dynamically loaded and unlinked, this may be the most convenient method. In many cases, it is practical to debug most of the trap handler routines in user state, then convert the trap module to system state. To make a trap handler execute in system state, you must set the supervisor state bit in the module attribute byte and create the module as super user. When the user trap executes, it is in system state.
- A program executes in system state if the supervisor state bit in the module's attribute word is set and the module is owned by the super user. This can be useful in rare instances.

IMPORTANT REMINDER: System-state routines are not time-sliced, therefore they should be as short and fast as possible.

Kernel System Call Processing

All OS-9 system calls (service requests) are processed through the kernel. The system-wide relocatable library files, sys.l and usr.l, define symbolic names for all system calls. The files are linked with hand-written assembly language or compiler-generated code. The OS-9 Assembler has a built-in macro to generate system calls:

```
OS9    I$Read
```

This is recognized and assembled to produce the same code as:

```
TRAP   #0
dc.w   I$Read
```

In addition, the C Compiler standard library includes functions to access nearly all user mode OS-9 system calls from C programs.

Parameters for system calls are usually passed and returned in registers. There are two general types of system calls: system function calls (calls that do not perform I/O) and I/O calls.

System Function Calls

There are two types of system function calls, user-state and system-state:

User-state System Calls

These requests perform memory management, multi-tasking, and other functions for user programs. They are mainly processed by the kernel.

System-state System Calls

Only system software in system state can use these calls, and they usually operate on internal OS-9 data structures. To preserve OS-9's modularity, these requests are system calls rather than subroutines. User-state programs cannot access them, but system modules such as device drivers may use them.

The symbolic name of each system function call begins with F$. For example, the system call to link a module is F$Link.

In general, system-state routines may use any of the user-state system calls. However, you must avoid making system calls at inappropriate times. For example, avoid I/O calls, timed sleep requests, and other calls that can be particularly time consuming (such as F$CRC) in an interrupt service routine.

Memory requested in system state is *not* recorded in the process descriptor memory list. Therefore, you must ensure that the memory is returned to the system before the process terminates.

WARNING: Avoid the F$TLink and F$Icpt system calls in system-state routines. Certain portions of the C library may be inappropriate for use in system state.

I/O Calls

I/O calls perform various I/O functions. The file manager, device driver, and kernel process I/O calls for a particular device. The symbolic names for this category of calls begin with I$. For example, the read service request is I$Read.

You may use any I/O system call in a system-state routine, with one slight difference than when executed in user-state. All path numbers used in system state are *system* path numbers. Each process descriptor has a path number that converts process local path numbers into system path numbers. The system itself has a global path number table to convert system path numbers into actual addresses of path descriptors. You must make system-state I/O system calls using system path numbers.

For example, the OS-9 F$PErr system call prints an error message on the caller's standard error path. To do this, it may not simply perform output on path number two. Instead it must examine the caller's process descriptor and extract the system path number from the third entry (0, 1, 2, ...) in the caller's path table.

When a user-state process exits with I/O paths open, the F$Exit routine automatically closes the paths. This is possible because OS-9 keeps track of the open paths in the process's path table. In system state, the I$Open and I$Create system calls return a system path number which is not recorded in the process path table or anywhere else by OS-9. Therefore, the system-state routine that opens any I/O paths must ensure that the paths are eventually closed, even if the underlying process is abnormally terminated.

Memory Management

To load any object (such as a program or constant table) into memory, the object *must* have the standard OS-9 memory module format as described in Chapter 1. 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.

OS-9 adds the module to the module directory when it is loaded into memory. 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 increments by one. When a process unlinks from a module, the module's link count decrements 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 attribute field (M$Attr) is set.

OS-9 Memory Map

OS-9 uses a software memory management system that contains all memory 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 2-1. Unless otherwise noted, the sections shown need not be located at specific addresses. However, Microware recommends that you keep each section in contiguous reserved blocks, arranged in an order that facilitates future expansion. Whenever possible, it is best to have physically contiguous RAM.

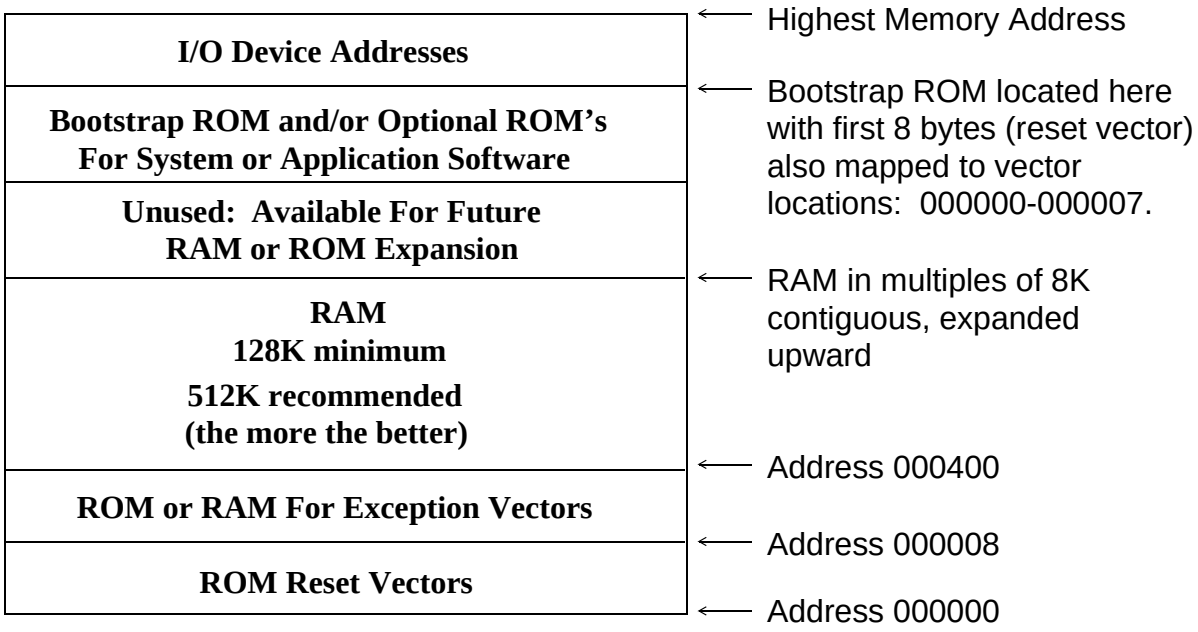


Figure 2-1: Typical OS-9 Memory Map

NOTE: For the 68020, 68030, 68040, and CPU32 family of CPUs, you can set the Vector Base Register (VBR) anywhere in the system. Thus, for these types of systems, there is no requirement that RAM or ROM be at address 0.

System Memory Allocation

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

OS-9 requires a variable amount of memory. Actual requirements depend on the system configuration and the number of active tasks and open files. The following sections describe approximate amounts of memory used by various parts of OS-9.

Operating System Object Code

A complete set of typical operating system component modules (kernel, file managers, device drivers, device descriptors, tick driver) occupies about 50K to 64K bytes of memory. On disk-based systems, these modules are normally bootstrap-loaded into RAM. OS-9 does not dynamically load overlays or swap system code; therefore, no additional RAM is required for system code.

You can place OS-9 in ROM for non-disk systems. The typical operating system object code for ROM-based, non-disk systems occupies about 30K to 40K bytes.

System Global Memory

OS-9 uses a minimum of 8K RAM for internal use. The system global memory area is usually located at the lowest RAM addressed. It contains an exception jump table, the debugger/boot variables, and a system global area. Variables in the system global area are symbolically defined in the sys.l library and the variable names begin with D_. The Reset SSP vector points to the system global area.

WARNING: User programs should *never* directly access system global variables because of issues such as portability and (depending on hardware) memory protection. System calls are provided to allow user programs to read the information in this area.

System Dynamic Memory

OS-9 maintains dynamic-sized data structures (such as I/O buffers, path descriptors, process descriptors, etc.) which are allocated from the general RAM area when needed. The System Global Memory area keeps pointers to the addresses of these data structures. A typical small system uses approximately 16K of RAM. The total depends on elements such as the number of active devices, the memory, and the number of active processes. The sys.l library source files include the exact sizes of all the system's data structure elements.

Free Memory Pool

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 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 to be run. Microware suggests that you have a system minimum of 128K plus an additional 64K per user available. Alternatively, small ROM-based control system might only need 32K 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 after they are loaded. This can lead to ***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. Refer to the following section for more information on colored memory.) If a number of program modules are loaded, and subsequently one or more non-contiguous modules are unlinked, several fragments of free memory space exist. The total free memory space may be quite large. However, because it is scattered, not enough space will exist in a single block to load a particular program module.

You can avoid memory fragmentation by loading modules at system startup. This places the modules in contiguous memory space. Or, you can initialize each standard device when booting the system. This allows the devices to allocate memory from higher RAM than would be available if the devices were initialized after booting.

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 to proceed. Use the mfree utility to determine the number and size of free memory blocks.

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 the F$Trans system call to perform 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

The colored memory list must end with a longword of zero. The following describes the MemList parameters:

Parameter	Size	Definition
Memory Type	word	Type of memory. Three memory types are currently defined in memory.h: SYSRAM 0x01 System RAM memory VIDEO1 0x80 Video memory for plane A VIDEO2 0x81 Video memory for plane B
Priority	word	Priority of memory (0-255). High priority memory is allocated first. If the block priority is 0, then the block can only be allocated by a request for the specific color (type) of the block.
Access permissions	word	Memory type access bit definitions: bit 0 B_USER User processes can allocate this memory. **NOTE:** This bit is ignored if the B_ROM bit is set. bit 1 B_PARITY Parity memory; the kernel initializes this memory during startup. **NOTE:** Only B_USER memory may be initialized. bit 2 B_ROM ROM; the kernel searches this memory for modules during startup. **NOTE:** B_ROM memory *cannot* be allocated by processes, as the B_USER and B_PARITY bits are ignored if B_ROM is set.
Search Block Size	word	The kernel checks every search block size to see if RAM/ROM exists.
Low Memory Limit	long	Beginning address of the block, as referenced by the CPU.
High Memory Limit	long	End address of the block, as referenced by the CPU.

Parameter	Size	Definition
Description String Offset	word	Offset of a user-defined string that describes the type of memory block.
Address Translation Adjustment	long	The external bus address of the beginning of the block. If zero, this field does not apply. Refer to F$Trans for more information.

The following is an example system memory map:

CPU Address	Bus Address	Memory Size	Physical Location
$00000000	**$00200000**	**$200000**	**on-board cpu ram**
$00600000	**$00600000**	**$200000**	**VMEbus ram**

A corresponding MemList table might appear as follows:

```
* memory list definitions for init module (user adjustable)
 align
* MemType  type, prior, attributes, blksiz, addr limits, name, DMA-offset
MemList
* on-board ram covered by "rom memory list:"
*  - this memory block is known to the "rom's memory list," thus it was
*    "parity initialized" by the rom code.
*  - the cpu's local base address of the block is at $00000000.
*  - the bus base address of the block is at $200000.
*  - this ram is fastest access for the cpu, so it has the highest priority.
*
 MemType SYSRAM,255,B_USER,4096,0,$200000,OnBoard,$200000

* off-board expansion ram
*  - this memory block is not known to the "rom's memory list,"
*    thus it needs "parity initialization" by the kernel.
*  - as the block is accessed over the bus, the base address of the block
*    is the same for cpu and dma accesses.
*  - this ram is slower access than on-board ram, therefore it
*    has a lower priority than the on-board ram.
*
 MemType SYSRAM,250,B_USER+B_PARITY,4096,$600000,$800000,OffBoard,0
 dc.l 0 end of list

OnBoard dc.b "fast on-board RAM",0
OffBoard dc.b "VMEbus memory",0
```

Colored Memory in Homogenous Memory Systems

Colored memory definitions are not essential for homogenous memory systems. However, colored memory definitions in this type of system can improve system performance and simplify memory list re-configuration.

System Performance

In a homogeneous memory system, the kernel allocates memory from the top of available RAM when requests are made by F$SRqMem (for example, when loading modules). If the system has RAM on-board the CPU and off-board in external memory boards, the modules tend to be loaded into the off-board RAM, because OS-9 always uses high memory first. On-board RAM is not used for a F$SRqMem call until the off-board memory is unable to accommodate the request.

Programs running in off-board memory execute slower than those running in on-board memory, due to bus access arbitration. Also, external bus activity increases. This may impact the performance of other bus masters in the system.

The colored memory lists can be used to reverse this tendency in the kernel, so that a CPU does not use off-board memory until all of its on-board memory is utilized. This results in faster program execution and less saturation of the system's external bus. Do this by making the priority of the on-board memory higher than off-board memory, as shown in the example lists on the preceding page.

Re-configuring Memory Areas

In a homogeneous memory system, the memory search areas are defined in the ROM's Memory List. If you do not use colored memory, you must make new ROMs from source code (usually impossible for end-users) or from a patched version of the original ROMs (usually difficult for end-users) to make changes to the memory search areas.

The colored memory lists simplify changes by configuring the search areas as follows:

- The ROM's memory list describes only the on-board memory.
- The colored memory lists in systype.d define the on-board memory and any external bus memory search areas in the Init module only.

The use of colored memory in a homogeneous memory system allows you to easily reconfigure the external bus search areas by adjusting the lists in systype.d and making a new Init module. The ROM does not require patching.

System Initialization

After a hardware reset, the bootstrap ROM executes the kernel (which is located in ROM or loaded from disk, depending on the system involved). The kernel initializes the system, which includes locating ROM modules and running the system startup task (usually Sysgo).

Init: The Configuration Module

Init is a non-executable module of type Systm (code $0C) which contains a table of system startup parameters. During startup, Init specifies initial table sizes and system device names, but it is always available to determine system limits. It must be in memory when the kernel executes and usually resides in the OS9Boot file or in ROM.

The Init module begins with a standard module header (Chapter 1, Figure 1-4) and the additional fields shown in the following table and in Figure 2-2.

NOTE: Refer to Appendix A for an example program listing of the Init module. Offset names are defined in the relocatable library sys.l.

Name	Description
M$PollSz	**IRQ polling size** The number of entries in the IRQ polling table. Each interrupt generating device control register requires one entry. The IRQ polling table has 32 entries by default. Each table entry is 18 bytes long.
M$DevCnt	**Device table size** The number of entries in the system device table. Each device on the system requires one entry in this table.
M$Procs	**Initial process table size** The initial number of active processes allowed in the system. If this table gets full, it automatically expands as needed.
M$Paths	**Initial path table size** The initial number of open paths in the system. If this table gets full, it automatically expand as needed.
M$SParam	**Offset to parameter string for startup module** The offset to the parameter string (if any) to pass to the first executable module.

Name	Description

M$SysGo **First executable module name offset**

The offset to the name string of the first executable module, usually Sysgo or Shell.

M$SysDev **Default directory name offset**

The offset to the initial default directory name string, usually /d0 or /h0. The kernel does a chd and chx to this device prior to forking the initial device. If the system does not use disks, this offset must be zero.

M$Consol **Initial I/O pathlist name offset**

The offset to the initial I/O pathlist string, usually /term. This pathlist is opened as the standard I/O path for the initial process. It is generally used to set up the initial I/O paths to and from a terminal. This offset should contain zero if no console device is in use.

M$Extens **Customization module name offset**

The offset to the name string of a list of customization modules (if any). A customization module complements or changes the existing OS-9 standard system calls. These modules are searched for at startup; they are usually found in the bootfile. If found, they execute in system state. Module names in the name string are separated by spaces. The default name string to search for is OS9P2. If there are no customization modules, set this value to zero.

NOTE: A customization module may only alter the d0, d1, and ccr registers.

NOTE: Customization modules must be system-type modules.

M$Clock **Clock module name offset**

The offset to the clock module name string. If there is no clock module name string, set this value to zero.

M$Slice **Time-slice**

The number of clock ticks per time-slice. If M$Slice is not specified, it defaults to 2.

M$Site **Offset to installation site code**

This value is usually set to zero. OS-9 does not currently use this field.

M$Instal **Offset to installation name**

The offset to the installation name string.

M$CPUTyp **CPU Type**

CPU type: 68000, 68008, 68010, 68020, 68030, 68040, 68070, 683xx.

Name	Description
M$OS9Lvl	**Level, version, and edition** This four byte field is divided into three parts: level: 1 byte version: 2 bytes edition: 1 byte For example, level 2, version 2.4, edition 0 would be 2240.
M$OS9Rev	**Revision offset** The offset to the OS-9 level/revision string.
M$SysPri	**Priority** The system priority at which the first module (usually Sysgo or Shell) executes. This is generally the base priority at which all processes start.
M$MinPty	**Minimum priority** The initial system minimum executable priority. For specific information on minimum priority, see the Process Execution section later in this chapter and F$SetSys in Chapter 1 of ***OS-9 System Calls***.
M$MaxAge	**Maximum age** The initial system maximum natural age. For specific information on maximum age, see the Process Execution section later in this chapter and F$SetSys in Chapter 1 of ***OS-9 System Calls***.
M$Events	**Number of entries in the events table** The initial number of entries allowed in the events table. If the table gets full, it automatically expands as needed. See the Events section of Chapter 3 for more specific information.

Name	Description

M$Compat **Revision compatibility**

This byte is used for revision compatibility. The following bits are currently defined:

Bit#	Function
0	Set to save all registers for IRQ routines
1	Set to prevent the kernel from using stop instructions
2	Set to ignore sticky bit in module headers
3	Set to disable cache burst operation (68030 systems)
4	Set to patternize memory when allocated or de-allocated
5	Set to prevent kernel cold-start from starting system clock

M$Compat2 **Revision compatibility #2**

This byte is used for revision compatibility. The following bits are currently defined**:**

Bit#		Function
0	0	External instruction cache is *not* snoopy*
	1	External instruction cache is snoopy or absent
1	0	External data cache is *not* snoopy
	1	External data cache is snoopy or absent
2	0	On-chip instruction cache is *not* snoopy
	1	On-chip instruction cache is snoopy or absent
3	0	On-chip data cache is *not* snoopy
	1	On-chip data cache is snoopy or absent
7	0	Kernel disables data caches when in I/O
	1	Kernel *does not* disable data caches when in I/O

* snoopy = cache that maintains its integrity without software intervention.

Name	Description

M$MemList **Colored memory list**

An offset to the memory segment list. The colored memory list contains an entry for each type of memory in the system. It is terminated by a long word of zero. See F$SRqCMem for further information. Each entry in the list has the following format:

Offset	Description
$00	Memory Type: SYSRAM = System RAM VIDEO1 = Plane A Video VIDEO2 = Plane B Video
$02	Priority
$04	Access permissions: B_USER = User processes allocate memory. B_PARITY = Parity memory; kernel initializes it during startup. B_ROM = ROM; kernel searches this for modules during startup.
$06	Search block size
$08	Low memory limit
$10	Offset to description string
$12	Reserved (must be zero)
$14	Address translation adjustment
$18	Reserved (must be zero)
$1C	Reserved (must be zero)

M$IRQStk **Kernel's IRQ stack size**

The size (in LONGWORDS) of the kernel's IRQ stack. The value of this field must be 0, or >= 256 and <=$ffff. If zero, the kernel uses a small default IRQ stack (not recommended).

M$ColdTrys **Retry counter**

This is the retry counter if the kernel's initial chd (to the default system device) fails.

NOTE: ***Offset*** refers to the location of a module field, relative to the starting address of the module. Module offsets are resolved in assembly code by using the names shown here and linking the module with the relocatable library: sys.l or usr.l.

Offset	Name	Description
$30	Reserved	Currently reserved for future use.
$34	M$PollSz	Number of IRQ polling table entries.
$36	M$DevCnt	Device table size.
$38	M$Procs	Initial process table size.
$3A	M$Paths	Initial path table size.
$3C	M$SParam	Parameter string for startup module (usually Sysgo).
$3E	M$SysGo	Offset to name string of first executable module.
$40	M$SysDev	Offset to the initial default directory name string.
$42	M$Consol	Offset to the initial I/O pathlist string.
$44	M$Extens	Offset to a name string of customization modules.
$46	M$Clock	Offset to the clock module name string.
$48	M$Slice	Number of clock ticks per time-slice.
$4A	Reserved	Currently reserved for future use.
$4C	M$Site	Offset to the installation site code.
$50	M$Instal	Offset to the installation name string.
$52	M$CPUTyp	CPU type.
$56	M$OS9Lvl	Level, version, and edition number of the operating system.
$5A	M$OS9Rev	Offset to the OS-9 level/revision string.
$5C	M$SysPri	Initial system priority.
$5E	M$MinPty	Initial system minimum executable priority.
$60	M$MaxAge	Initial system maximum natural age.
$62	Reserved	Currently reserved for future use.
$66	M$Events	Initial number of entries allowed in the events table.
$68	M$Compat	Compatibility flag one. Byte is used for revision compatibility.
$69	M$Compat2	Compatibility flag two. Byte is used for revision compatibility.
$6A	M$MemList	Offset to the memory segment list.
$6C	M$IRQStk	Size of the kernel's IRQ stack.
$6E	M$ColdTrys	Retry counter if the kernel's initial chd fails.

NOTE: The strings themselves follow the 28 byte reserved section.

Figure 2-2: Additional Fields for the Init Module

Sysgo

Sysgo is the first user process started after the system startup sequence. Its standard I/O is on the system console device.

Sysgo usually executes as follows:

- ¿ Changes to the CMDS execution directory on the system device.
- ¡ Executes the startup file (as a script) from the root of the system device.
- ¬ Forks a shell on the system console.
- Ð Waits for that shell to terminate and then forks it again. Therefore, there is always a shell running on the system console, unless Sysgo dies.

You cannot use the standard Sysgo module for disk systems on non-disk systems, but it is easy to customize.

You may eliminate Sysgo by specifying shell as the initial startup module and specifying a parameter string similar to:

```
startup; ex tsmon /term
```

See Appendix A for an example source listing of the Sysgo module.

Customization Modules

Customization modules are additional modules you can execute at boot time to enhance OS-9's capabilities. They provide a convenient way to install a new system call code or collection of system call codes, for example, a system security module. The kernel calls the modules at boot time if their names are specified in the extension list of the Init module and the kernel can locate them.

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

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

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

¡ 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, add the following line immediately before the endm line:

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

NOTE: os9p2 is the name of the default customization module.

Remake the Init module.

¬ Create a new bootfile:

Change to the /h0/CMDS/BOOTOBJS directory and edit the bootlist file so that the customization module name appears in the list.

Create a new bootfile with the os9gen utility. For example:

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

Ð Reboot the system and make sure that the new module is operational.

Process Creation

All OS-9 programs run as ***processes*** or ***tasks***. The F$Fork system call creates new processes. The name of the primary module that the new process is to execute initially is the most important parameter passed in the fork system call. The following outlines the creation process:

¿ ***Locate or Load the Program.***
OS-9 tries to find the module in memory. If it cannot find the module, it loads a mass-storage file into memory using the requested module name as a file name.

¡ ***Allocate and Initialize a Process Descriptor.***
After OS-9 locates the primary module, it assigns a data structure called a ***process descriptor*** to the new process. The process descriptor is a table that contains information about the process: its state, memory allocation, priority, I/O paths, etc. The process descriptor is automatically initialized and maintained. The process does not need to know about the descriptor's existence or contents.

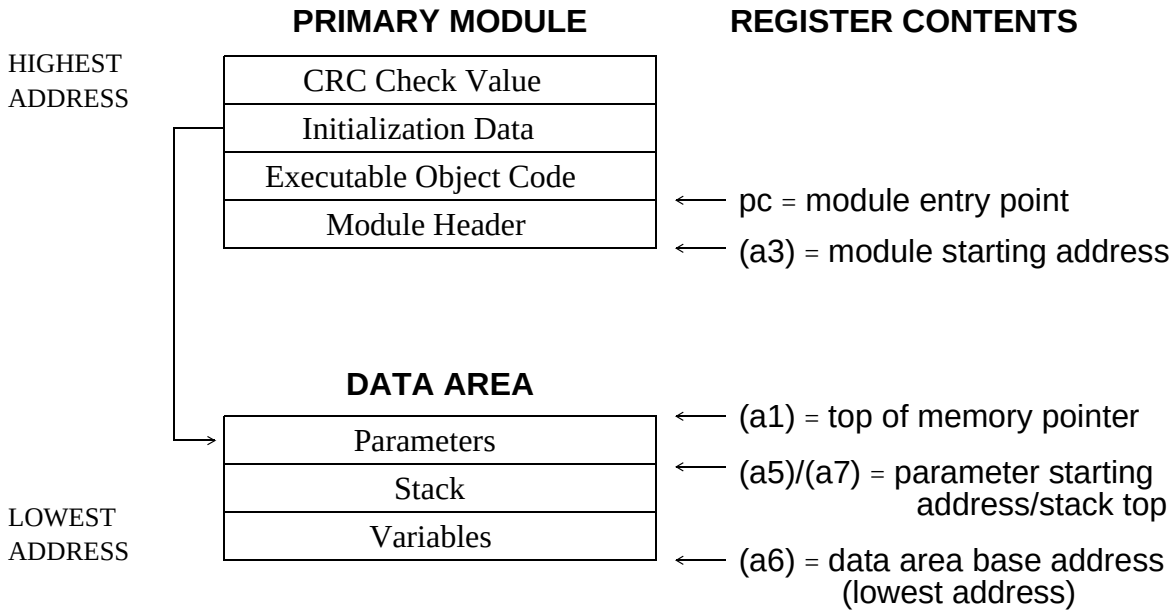
¬ ***Allocate the Stack and Data Areas.***
The primary module's header contains a data and stack size. OS-9 allocates a ***contiguous memory area*** of the required size from the free memory space. The following section discusses process memory areas.

Ð ***Initialize the Process.***
OS-9 sets the new process's registers to the proper addresses in the data area and object code module (see Figure 2-3). 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.

If OS-9 cannot perform any of these steps, it aborts the creation of the new process and notifies the process that originated the fork of the error. If OS-9 completes all the steps, it adds the new process to the active process queue for execution scheduling.

The new process is also assigned a ***process ID***. This is a unique number which is the process's identifier. Other processes can communicate with it by referring to its ID in system calls. The process also has an associated ***group ID*** and ***user ID***. These identify all processes and files belonging to a particular user and group of users. The group and user ID's are inherited from the parent process.

Processes terminate when they execute an F$Exit system service request or when they receive fatal signals or errors. Terminating the process closes any open paths, de-allocates the process's memory, and unlinks its primary module.



Registers passed to the new process:

sr	0000	(a0)	undefined
pc	module entry point	(a1)	top of memory pointer
d0.w	process ID	(a2)	undefined
d1.l	group/user ID	(a3)	primary module pointer
d2.w	priority	(a4)	undefined
d3.w	# of paths inherited	(a5)	parameter pointer
d4.l	undefined	(a6)	static storage (data area) base pointer
d5.l	parameter size	(a7)	stack pointer (same as a5)
d6.l	total initial memory allocation		
d7.l	undefined		

NOTE: (a6) is always biased by $8000 to allow object programs to access 64K of data using indexed addressing. You can usually ignore this bias because the OS-9 linker automatically adjusts for it.

Figure 2-3: New Process's Initial Memory Map And Register Contents

Process Memory Areas

OS-9 divides all processes (programs) into two logically separate memory areas: code and data. This division provides OS-9's modular software capabilities.

Each process has a unique data area, but not necessarily a unique program memory module. This allows two or more processes to share the same copy of a program. This technique is an automatic function of OS-9 and results in extremely efficient use of available memory.

A program must be in the form of an executable memory module (described in Chapter 1) to be run. The program is position-independent and ROMable, and the memory it occupies is considered read-only. It may link to and execute code in other modules.

The process's data area is a separate memory space where all of the program's variables are kept. The top part of this area is used for the program's stack. The actual memory addresses assigned to the data area are unknown at the time the program is written. A base address is kept in a register (usually a6) to access the data area. You can read or write to this area.

If a program uses variables that require initialization, OS-9 copies the initial values from the read-only program area to the data area where the variables actually reside. The OS-9 linker builds appropriate initialization tables which initialize the variables.

Process States

A process is either in active, waiting, or sleeping state:

ACTIVE The process is active and ready for execution. The scheduler gives active processes time for execution according to their relative priority with respect to all other active processes. It uses a method that compares the ages of all active processes in the queue. It gives some CPU time to all active processes, even if they have a very low relative priority.

WAITING The process is inactive until a child process terminates or until a signal is received. The process enters the wait state when it executes a F$Wait system service request. It remains inactive until one of its descendant processes terminates or until it receives a signal.

SLEEPING The process is inactive for a specified period of time or until it receives a signal. A process enters the sleep state when it executes an F$Sleep service request. F$Sleep 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.

There is a separate queue (linked list of process descriptors) for each process state. State changes are made by moving a process descriptor from its current queue to another queue.

Process Scheduling

OS-9 is a multi-tasking operating system, that is, two or more independent programs, called ***processes*** or ***tasks***, can execute 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, and how long, to run based on the priorities of the active processes. ***Task-switching*** is the action of switching from the execution of one process to another. Task-switching does not affect the programs' execution.

A real-time clock interrupts the CPU at 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. Thus, to change the tick length, you must rewrite the clock driver and re-initialize the hardware.

A ***slice*** or ***time-slice*** is the longest amount of time a process will control the CPU before the kernel re-evaluates the active process queue. By default, a slice is two ticks. You can change the number of ticks per slice by adjusting the system global variable D_TSlice or by modifying the Init module.

To ensure efficiency, only processes on the active process queue are considered for execution. The active process queue is organized by ***process age***, a count of how many task switches have occurred since the process entered the active queue plus the process's initial priority. The oldest process is at the head of the queue. OS-9's scheduling algorithm allocates some execution time to each active process.

When a process is placed in the active queue, its age is set to the process's assigned priority and the ages of all other processes increment. Ages never increment beyond $FFFF.

After the currently executing process's time-slice, the kernel executes the process with the highest age.

Pre-emptive Task-switching

During critical real-time applications you sometimes need fast interrupt response time. 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.

Task-switching is affected by two system global variables: D_MinPty (minimum priority) and D_MaxAge (maximum age). Both variables are initially set in the Init module. Users with a group ID of zero (super users) can access both variables through the F$SetSys system call.

If a process's priority or age is less than D_MinPty, the process is not considered for execution and is not aged. Usually, this variable is not used; it is set to zero.

WARNING: If the minimum system priority is set above the priority of all running tasks, the system is completely shut down. You must reset to recover. Therefore, it is crucial to restore D_MinPty to a normal level when the critical task(s) finishes.

D_MaxAge is the maximum age to which a process can increment. When D_MaxAge is activated, tasks are divided into two classes, high priority and low priority:

- High priority tasks receive all of the available CPU time and do not age.
- Low priority tasks do not age past D_MaxAge. Low priority tasks are run only when the high priority tasks are inactive. Usually, this variable is not used; it is set to zero.

NOTE: A system-state process is *not* pre-empted until it finishes, unless it voluntarily gives up its time-slice. This exception is made because system-state processes may be executing critical routines that affect shared system resources which may block other unrelated processes.

Exception and Interrupt Processing

One of OS-9's features is its extensive support of the 68K family advanced exception/interrupt system. You can install routines to handle particular exceptions using various OS-9 system calls for the types of exceptions.

Vector Number	Related OS-9 Call	Assignment
0	none	Reset initial Supervisor Stack Pointer (SSP)
1	none	Reset initial Program Counter (PC)
2	F$STrap	Bus error
3	F$STrap	Address error
4	F$STrap	Illegal instruction
5	F$STrap	Zero divide
6	F$STrap	CHK instruction; CHK2 (CPU32)
7	F$STrap	TRAPV instruction
8	F$STrap	Privilege violation
9	F$DFork	Trace
10	F$STrap	Line 1010 emulator
11	F$STrap	Line 1111 emulator
12	none	Reserved (000/008/010/070); hardware break point (CPU32)
13	none	Reserved
14	none	Reserved (000/008); format error (010/070/CPU32)
15	none	Uninitialized interrupt
16-23	none	Reserved
24	none	Spurious interrupt
25-31	F$IRQ	Level 1-7 interrupt autovectors
32	F$OS9	User TRAP #0 instruction (OS-9 call)
33-47	F$TLink	User TRAP #1-15 instruction vectors
48-56	none	Reserved
57-63	none/F$IRQ	Reserved (000/008/010/CPU32) on-chip level 1-7 auto-vectored interrupts (070)
64-255	F$IRQ	Vectored interrupts (user defined)

Figure 2-4: Vector Descriptions for 68000/008/010/070/CPU32 Family

Vector Number	Related OS-9 Call	Assignment
0	none	Reset initial Supervisor Stack Pointer (SSP)
1	none	Reset initial Program Counter (PC)
2	F$STrap	Bus error
3	F$STrap	Address error
4	F$STrap	Illegal instruction
5	F$STrap	Zero divide
6	F$STrap	CHK, CHK2
7	F$STrap	TRAPV cpTRAPcc, TRAPcc
8	F$STrap	Privilege violation
9	F$DFork	Trace
10	F$STrap	Line 1010 emulator
11	F$STrap	Line 1111 emulator
12	none	Reserved
13	none	Coprocessor protocol violation (020,030 only); reserved (040)
14	none	Format error
15	none	Uninitialized interrupt
16-23	none	Reserved
24	none	Spurious interrupt
25-31	F$IRQ	Level 1-7 interrupt autovectors
32	F$OS9	User TRAP #0 instruction (OS-9 call)
33-47	F$TLink	User TRAP #1-15 instruction vectors
48	F$STrap	FPCP Branch, or set on unordered condition
49	F$STrap	FPCP Inexact result
50	F$STrap	FPCP Divide by zero
51	F$STrap	FPCP Underflow error
52	F$STrap	FPCP Operand error
53	F$STrap	FPCP Overflow error
54	F$STrap	FPCP NAN signaled
55	F$STrap	Reserved (020/030); FPCP Unimplemented data type (040)
56	none	PMMU Configuration (020/030); reserved (040)
57	none	PMMU Illegal Operation (020); reserved (030/040)

58	none	PMMU Access Level Violation (020); reserved (030/040)
59-63	none	Reserved
64-255	F$IRQ	Vectored interrupts (user defined)

Figure 2-5: Vector Descriptions for 68020/030/040

Reset Vectors: vectors 0, 1

The reset initial SSP vector contains the address loaded into the system's stack pointer at startup. There must be at least 4K of RAM below and 4K of RAM above this address for system global storage. Each time an exception occurs, OS-9 uses this vector to find the base address of system global data.

The reset initial program counter (PC) is the coldstart entry point to OS-9. After startup, its only use is to reset after a catastrophic failure.

WARNING: User programs should not use or modify either of these vectors.

Error Exceptions: vectors 2-8, 10-24, 48-63

These exceptions are usually considered fatal program errors and cause a user program to unconditionally terminate. If F$DFork created the process, the process resources remain intact and control returns to the parent debugger to allow a postmortem examination.

You may use the F$STrap system call to install a user subroutine to catch the errors in this group that are considered non-fatal.

When an error exception occurs, the user subroutine executes in user state, with a pointer to the normal data space used by the process and all user registers stacked. The exception handler must decide whether and where to continue execution.

If any of these exceptions occur in system state, it usually means a system call was passed bad data and an error is returned. In some cases, system data structures are damaged by passing nonsense parameters to system calls.

NOTE: Not all catchable exception vectors are applicable to all 68000-family CPUs. For example, vectors 48-54 (FPCP exceptions) only apply to 68020 and 68030 CPUs.

The Trace Exception: vector 9

The trace exception occurs when the status register trace bit is set. This allows the MPU to single step instructions. OS-9 provides the F$DFork, F$DExec, and F$DExit system calls to control program tracing.

AutoVectored Interrupts: vectors 25-31; 57-63 (68070 only)

These exceptions provide interrupt polling for I/O devices that do not generate vectored interrupts. Internally, they are handled exactly like vectored interrupts (see below).

WARNING: Normally, you should not use Level 7 interrupts because they are non-maskable and can interrupt the system at dangerous times. You may use Level 7 interrupts for software refresh of dynamic RAMs or similar functions, provided that the IRQ service routine does not use any OS-9 system calls or system data structures.

User Traps: vectors 32-47

The system reserves user trap zero (vector 32) for standard OS-9 system service requests. The remaining 15 user traps provide a method to link to common library routines at execution time.

Library routines are similar to program object code modules and are allocated their own static storage when installed by the F$TLink service request. The execution entry point executes whenever the user trap is called. In addition, trap handlers have initialization and termination entry points which execute when linked and at process termination. The termination entry point is not currently implemented.

NOTE: Trap 13 (CIO) and trap 15 (math) are standard trap handlers distributed by Microware.

Vectored Interrupts: vectors 64-255

The 192 vectored interrupts provide a minimum amount of system overhead in calling a device driver module to handle an interrupt. Interrupt service routines execute in system state without an associated current process. The device driver must provide an error entry point for the system to execute if any error exceptions occur during interrupt processing, although this entry point is not currently implemented. The F$IRQ system call installs a handler in the system's interrupt tables. If necessary, multiple devices may be used on the same vector.

End of Chapter 2