CHAPTER 11

OS-9 SYSTEM CALLS

All operating system functions are accessed by means of system calls. A system call is essentially a subroutine that is executed in system state. It is important to appreciate that the system call is effectively a subroutine call by the calling program, so that all operations performed by the system call handler function are performed on behalf of the calling process. For example, if a system call function goes to sleep, it is the calling process that is sleeping. Similarly, the calling program will not continue execution until the system call is completed, just as would be the case with a simple subroutine call.

Conversely, the operating system does nothing without a system call being made by a program. The only time that operating system code executes other than when servicing a system call is when servicing an interrupt.

11.1 THE SYSTEM CALL MECHANISM

OS-9 system calls use the 68000 family **trap #0** instruction. This instruction - an extension of the software interrupt of earlier processors - causes a processor exception. The processor saves the status register and program counter, switches to supervisor state, and continues execution at the address indicated by the appropriate exception vector. There are 16 **trap** instructions, **trap #0** to **trap #15**. OS-9 uses **trap #0** for system calls.

The required system function is indicated by the word following the **trap #0** instruction in the program. The kernel reads this word to determine the required function, using it as an index into the appropriate Dispatch Table (System or User) depending on whether the caller was in system or user state. The kernel then adds two to the saved program counter, so that

program execution will continue with the instruction following the function code word.

The OS-9 assemblers **r68** and **r68020** provide a built-in macro **os9** to generate the **trap #0** instruction and the function code word in one statement. The function codes are defined symbolically in the file 'DEFS/funcs.a', and so are available as external references resolved by the library 'LIB/sys.l'. For example, the "fork a process" system call has function code 3, defined symbolically as **F$Fork**. So the assembly language statement:

```
os9     F$Fork
```

is equivalent to the statements:

```
trap    #0
dc.w    F$Fork
```

or:

```
trap    #0
dc.w    3
```

The **trap** instructions, as with any exception, put the processor in supervisor state, which is the state required by the operating system. Note that the 68000 family processors can only go from user state to supervisor state as the result of an exception.

11.2 SYSTEM CALL PARAMETERS

The **trap #0** exception handler in the kernel saves all of the caller's registers on the system stack[13] as a "stack frame", and retrieves them before returning to the caller. Therefore the caller's registers are not modified unless explicitly stated in the documentation, because the system call function must actively change the stack frame to affect the caller's registers.

OS-9 was originally written exclusively in assembly language (although now some parts are written in C), with speed of execution in mind. Therefore all parameters to system calls, and all returned values, are in processor registers, or are pointed to by processor registers. Which registers hold which values is determined by the specification for the particular system call.

The error returning convention is the same for all system calls, and for all communications between operating system components. The kernel returns an error to the calling process in the caller's registers - the "carry" flag of the caller's condition codes register (**ccr**) is set, and the error code is placed in

13 The system stack is in the second half of the calling process's process descriptor, except during interrupt processing.

the low word (bits 0-15) of the **d1** register. If there is no error, the carry flag is cleared, and the **d1** register is not modified.

The passing of parameters and results in the processor registers and the setting of the carry flag to indicate an error are techniques that are not directly compatible with C. Therefore it is necessary to provide small library functions written in assembly language that can be called from C programs. Such functions translate the C parameter passing format to that required by the system call, make the system call, and then convert the returned values and error indication to a form compatible with C. This topic is covered in detail in the chapter on Microware C and Assembly Language.

11.3 CUSTOM SYSTEM CALLS

System state modules can add new system calls, or replace existing ones, using the **F$SSvc** system call. This makes OS-9 almost infinitely customizable. Note that this is a privileged system call - normal user state programs cannot add or modify system calls.

On coldstart the kernel installs its own system call handlers in the Dispatch Tables, using the **F$SSvc** system call routine. The kernel then tries to link to one or more "kernel customization modules", whose names are given in the **init** module. If found, the kernel calls their entry points, giving them a chance to allocate memory, set up data structures, and install system calls using **F$SSvc**. The system programmer can therefore add or modify system calls without altering the kernel.

New system calls can make themselves extensions to existing system calls by saving the address of the existing routine (from the Dispatch Tables) before installing the new routine. When called, the new routine performs its additional function, and then jumps to the old routine (or calls the old routine first, as appropriate).

The **F$SSvc** system call optionally instructs the kernel to save a memory address in the Dispatch Table with each system call handler address. A kernel customization module can include static storage definitions, in the same way as a program. If it does, the **M$Mem** field of its module header gives the total size of the required static storage. Before calling the initialization routine of a kernel customization module, the kernel allocates an area of memory of the required size, and passes its address to the customization module's initialization routine in the **a3** register.

The **F$SSvc** system call expects the "memory address" parameter to be in the **a3** register, so when the initialization routine makes this system call to install a new or replacement system call, the module's static storage pointer is automatically saved in the Dispatch Table. When the kernel calls the new system call handler routine, it passes the saved memory address in the **a3** register, permitting the system call handler to access the static storage using the symbolic names with which the static storage was defined. This effectively allows the System Globals memory structure to be extended as needed.

11.4 USER AND SYSTEM STATE CALLS

There are two Dispatch Tables - a user table and a system table. When a system call is made the kernel uses one or the other table to fetch the appropriate routine address (indexed by the system call code). The choice is made on the basis of the processor state of the caller, determined by inspecting the saved status register on the stack. Each table is 512 long words. The first group of 256 entries are the addresses of the system call handler functions, indexed by the system call code. The second group of 256 entries are the memory addresses appropriate to the system calls (set by the **F$SSvc** call), also indexed by the system call code.

The **F$SSvc** system call takes a flag with each call being installed indicating whether the routine address should be put in both dispatch tables, or in the System Dispatch Table only (for privileged calls). A function can be made to behave differently depending on whether the call was made from user or system state by installing the user state version of the routine in both tables, and then the system state version of the routine in the System table only.

This is commonly done by the kernel for I/O system calls. A user state call has its path number translated through the path number conversion table in its process descriptor, to give the system path number identifying the appropriate path descriptor. A system state call passes the system path number directly.

11.5 THE SYSTEM CALLS

Each system call provided by OS-9 version 2.4 is listed below, with its function code and a brief description of its purpose. Many of the function codes are historical, and are no longer used. Others have been defined for future use, or for special applications. These unimplemented codes are shown with the description in italics. Privileged system calls are shown with a ✱

symbol preceding the description. System calls that only apply if the System Security Module is in use are shown with "SSM" preceding the description.

The I/O system calls are described in detail in the section on the I/O System.

Some system calls were created by Microware specifically for use in their utilities, and are not documented in the OS-9 Technical Manual, while other system calls were not documented until recently. These calls are shown with a ▪ after the description, and are briefly described in the sections following the table below.

Code	Name	Description
$000	F$Link	Link to a module in memory.
$001	F$Load	Read a file of one or more modules into memory and install the modules as a group in the module directory. ▪
$002	F$UnLink	Unlink a module, specifying the module address. From OS-9 version 2.3 onwards, if the call is from user state and the SSM is in use, the module header must be in the caller's memory map. If the unlink reduces the link count to zero, (or -1 for a sticky module), the kernel deletes the module from the module directory, subject to certain checks that the module is not in use. A module whose type code is greater than 12 is assumed to be an I/O module, and the kernel makes an **F$IODel** system call to check that it is not in use by an I/O sub-system. From OS-9 version 2.4.3 (released in 1992), the kernel checks each process descriptor to ensure that the module is not a primary program module or an installed trap handler.
$003	F$Fork	Start a process, specifying a program module to link to or file name to load. The kernel first attempts to link to a module of the given name. If that fails, the kernel attempts to load from a file of the given name, relative to the execution directory. If this succeeds, the kernel will execute the module loaded (irrespective of the name). If the file contains more than one module, the first module is executed. The link and load requests are executed on behalf of the new process, not the parent process.
$004	F$Wait	Wait for any child of this process to die.
$005	F$Chain	Convert this process to executing a new program module.
$006	F$Exit	Terminate this process.
$007	F$Mem	Change the size of the primary data area (initial static storage and stack) of this process (not recommended for new applications). This call will fail if it attempts to expand the static storage, and insufficient contiguous free memory is available above the current static storage. The **F$Fork** system call uses this system call to allocate the primary data area for the process. Memory is allocated from low memory upwards (contrast **F$SRqMem**).
$008	F$Send	Send a signal to a process.

$009	F$Icpt	Install or replace the signal handler function for this process. A handler address of zero cancels any currently installed signal handler for the process.
$00A	F$Sleep	Put this process to sleep for a time, or until woken by a signal. A request for a sleep of one tick immediately re-inserts the process in the active queue, causing a reschedule. A request for a sleep of **n** ticks will sleep until **n-1** tick interrupts are received.
$00B	F$SSpd	*(Suspend Process).*
$00C	F$ID	Return the process ID of this process.
$00D	F$SPrior	Set the execution priority of a process.
$00E	F$STrap	Install a handler function for "hardware" exceptions (bus error, address error, illegal instruction, and so on).
$00F	F$PErr	Print an error number, with an optional description string searched for in a text file.
$010	F$PrsNam	Parse the name of a module or file.
$011	F$CmpNam	Compare a match string with a file or module name, including wild card characters.
$012	F$SchBit	Search a bit map for a free (clear) field.
$013	F$AllBit	Allocate (set) a field in a bit map.
$014	F$DelBit	De-allocate (clear) a field in a bit map.
$015	F$Time	Get the current data and time.
$016	F$STime	Set a new date and time, or read the date and time from a battery-backed clock (by specifying a date of zero).
$017	F$CRC	Generate or check the module CRC over part or all of a module (or other memory area).
$018	F$GPrDsc	Get a copy of the process descriptor of a process.
$019	F$GBlkMp	Get information about the free memory list.
$01A	F$GModDr	Get a copy of the module directory.
$01B	F$CpyMem	Copy from an absolute memory address to the caller's buffer (the process ID parameter mentioned in the OS-9 Technical Manual is not used).
$01C	F$SUser	Change the user and group numbers of this process. Only permitted in two cases. Firstly, if the caller is a super user (group 0). Or secondly, if the owner of the primary module (in the **M$Owner** field of the program module header) is a super user and was at the time of forking (the kernel compares the **M$Owner** field with the **P$MOwn** field of the process descriptor), and the new user and group are to be the same as the module owner. Note: prior to OS-9 version 2.3 the check that **M$Owner** had not changed was omitted.
$01D	F$UnLoad	Unlink a module by name.

$01E	F$RTE	Exit a signal handler function (this call must be used to ensure that all pending signals are processed).
$01F	F$GPrDBT	Get a copy of the process descriptor table.
$020	F$Julian	Convert a date and time in Gregorian format to Julian format.
$021	F$TLink	Link to a trap handler module and install it to handle future **trap #n** instructions from this process.
$022	F$DFork	Fork a process to be debugged.
$023	F$DExec	Execute one or more instructions of a child process being debugged.
$024	F$DExit	Terminate a child process being debugged.
$025	F$DatMod	Create a data module (or other module type) in memory.
$026	F$SetCRC	Correct the header parity and CRC of a module in memory.
$027	F$SetSys	Read or write a field of the System Globals.
$028	F$SRqMem	Allocate memory by priority only (no regard for colour). This call will not allocate memory of priority zero.
$029	F$SRtMem	De-allocate memory (return it to the free pool).
$02A	F$IRQ	Install an interrupt handler function in the interrupt polling table.
$02B	F$IOQu	I/O queue this process on another process (that is using an I/O resource). The queue is ordered by the scheduling constants of the processes at the time they were placed in the queue. A process being added to the queue is placed later in the queue than other processes in the queue with equal or greater scheduling constants.
$02C	F$AProc	Put a process in the active queue.
$02D	F$NProc	Make the first process in the active queue the current process.
$02E	F$VModul	Validate a module in memory and install it in the module directory.
$02F	F$FindPD	Get the address of a path or process descriptor, given the path number or process ID and the base address of the path or process descriptor table.
$030	F$AllPD	Allocate a new path or process descriptor, given the base address of the path or process descriptor table. Searches the table for a free entry (which determines the new path number or process ID), allocates the required memory and clears it, and sets the address in the table. Then sets the first word of the allocated memory to the path number or process ID, and returns the path number or process ID and the address of the allocated memory. The first two words of the table give information about the memory structures. The first word is the current maximum path number or process ID permitted (equal to the size of the table in long words, minus one). The second word is the size of each structure (path or process descriptor).

$031	F$RetPD	※ De-allocate a path or process descriptor, given a path number or process ID in **d0** and the base address of the path or process descriptor table in **a0**. Clears the table entry for this descriptor, and de-allocates the memory.
$032	F$SSvc	※ Install one or more system call handler routines.
$033	F$IODel	※ Check unlinking of an I/O module (file manager, device driver or device descriptor). ▮
$037	F$GProcP	※ Get the address of a process descriptor given a process ID. ▮
$038	F$Move	※ Optimized memory copy (also takes into account any MMU restrictions).
$039	F$AllRAM	*(Allocate RAM blocks).*
$03A	F$Permit	SSM - Add memory area to process's memory map (permits the process to access the memory area). ▮
$03B	F$Protect	SSM - Remove memory area from process's memory map. ▮
$03C	F$SetImg	*(Set Process DAT Image).*
$03D	F$FreeLB	*(Get Free Low Block)*
$03E	F$FreeHB	*(Get Free High Block)*
$03F	F$AllTsk	※ SSM - Ensure the MMU is set up for this process. If the SSM is not installed, this call is not privileged - it does nothing, and returns no error. ▮
$040	F$DelTsk	※ SSM - De-allocate the task number for this process. If the SSM is not installed, this call is not privileged - it does nothing, and returns no error. ▮
$041	F$SetTsk	*(Set Process Task DAT registers).*
$042	F$ResTsk	*(Reserve Task number).*
$043	F$RelTsk	*(Release Task number).*
$044	F$DATLog	*(Convert DAT Block/Offset to Logical).*
$045	F$DATTmp	*(Make temporary DAT image).*
$046	F$LDAXY	*(Load A [X,[Y]]).*
$047	F$LDAXYP	*(Load A [X+,[Y]]).*
$048	F$LDDDXY	*(Load D [D+X,[Y]]).*
$049	F$LDABX	*(Load A from 0,X in task B).*
$04A	F$STABX	*(Store A at 0,X in task B).*
$04B	F$AllPrc	※ Allocate a new process descriptor - calls **F$AllPD**, then sets the current date and time in the **P$DatBeg** and **P$TimBeg** fields of the process descriptor.
$04C	F$DelPrc	※ De-allocate a process descriptor, given the process ID in the **d0** register. Calls **F$RetPD**.
$04D	F$ELink	*(Link using Module Directory Entry).*

$04E	F$FModul	Find a module directory entry.
$04F	F$MapBlk	*(Map Specific Block).*
$050	F$ClrBlk	*(Clear Specific Block).*
$051	F$DelRAM	*(De-allocate RAM blocks).*
$052	F$SysDbg	Invoke system level debugger.
$053	F$Event	Create, link to, unlink from, delete, change, inspect, or wait for an OS-9 event.
$054	F$Gregor	Convert a date and time in Julian format to Gregorian format.
$055	F$SysID	Get the system identification information.
$056	F$Alarm	Set up to be sent a signal after a timed interval, or periodically. In system state, install a handler function to be called after a timed interval, or periodically.
$057	F$SigMask	Increment, decrement, or clear the signal mask for this process. The **d0** register must be zero. The **d1** register must be -1 (to decrement the signal mask), or 0 (to clear the signal mask), or 1 (to increment the signal mask).
$058	F$ChkMem	SSM - check that a memory area is within this process's memory map. If "write" permission is requested, the SSM checks that the memory is not write protected from this process. Otherwise, it checks that the process can read and execute the memory. In user state, or if SSM is not used, this call just reads the first byte of the memory area (generating a bus error if the memory is not accessible).
$059	F$UAcct	For a user accounting module. The kernel makes this call when a process is forked, chained, or terminated. A kernel customization module can install a handler for this call, and maintain user accounting information.
$05A	F$CCtl	Enable, disable, or flush the processor program and/or data caches.
$05B	F$GSPUMp	SSM - get a copy of the memory map of a process.
$05C	F$SRqCMem	Allocate memory of a particular colour. This call will allocate memory of priority zero if no other memory of that colour is available.
$05D	F$POSK	*(Execute service request).*
$05E	F$Panic	Panic warning. The kernel has no handler for this call, but makes this call if all processes have been terminated. Custom modules could make this call under other fatal conditions, such as power failure. A watchdog module could install a handler for this call to handle these situations gracefully. Normally this call would be installed for system state use only.
$05F	F$MBuf	*(Memory buffer manager). This system call is implemented as part of the Internet Support Package (ISP).*

$060	F$Trans	Translate a memory address as seen by the CPU into the address to be used by an alternate bus master, using the address translation offset given in the memory list in the **init** configuration module.
$080	I$Attach	Ensure an I/O device is initialized.
$081	I$Detach	Terminate usage of an I/O device.
$082	I$Dup	Get another local path number for an open path.
$083	I$Create	Open a path and create a file.
$084	I$Open	Open a path to an existing file or device.
$085	I$MakDir	Create a directory file.
$086	I$ChgDir	Change the current data and/or execution directory for this process.
$087	I$Delete	Delete a file.
$088	I$Seek	Change the current file pointer on a path.
$089	I$Read	Read from a path without data editing.
$08A	I$Write	Write to a path without data editing.
$08B	I$ReadLn	Read from a path, terminating on [CR], allowing data editing.
$08C	I$WritLn	Write to a path, terminating on [CR], allowing data editing.
$08D	I$GetStt	Get information about a path, file, or device.
$08E	I$SetStt	Modify information or operation, or request special action, of a path, file, or device.
$08F	I$Close	Close a path.
$092	I$SGetSt	Get a copy of the device name or path descriptor options section of an open path using a system path number.

11.5.1 F$AllTsk System Call

The kernel makes this system call just before starting or restarting a process in user state. It is a request to the SSM to ensure the MMU is correctly set up for the current process. Some MMUs can store multiple process memory maps simultaneously. The current map is selected by writing a number to the MMU. Under OS-9 this number is known as a task number.

If the MMU can store multiple maps the SSM first checks to see whether the map for this process is already in the MMU - that is, a task number is allocated to the process. In this case the SSM need only write the task number to the appropriate MMU register to select the map for the current process. The SSM stores the process's task number in the **P$Task** field of the process descriptor. (If the process's map is not currently in the MMU the SSM sets **P$Task** to some invalid value as an indication of this.) If no task

number is currently allocated to the current process the SSM tries to find an unallocated task number for it. Otherwise it must take a task number from another process.

In the case that the MMU cannot store multiple maps the SSM will keep a record of the process descriptor address of the process whose map is currently in the MMU, so that it does not unnecessarily rewrite the map in the MMU.

The SSM will therefore have decided whether the memory map for the current process must be written to the MMU. Some MMUs can read the processor's memory, so that they read the map themselves as necessary. For these MMUs (such as the MMUs in the 68030 and 68040) the SSM only needs to write the root address of the process's memory map to the MMU register. Otherwise the SSM must copy the process's memory map to the MMU internal memory.

The SSM will also need to copy the process's memory map to the MMU internal memory if the process's memory map has changed - the map previously stored in the MMU is "stale". The SSM **F$Protect** and **F$Permit** system calls set bit 4 of the **P$State** field in the process's process descriptor to indicate that the memory map of the process has been changed. If the MMU uses internal memory to store the map (rather than directly accessing the processor's memory), the **F$AllTsk** system call must update the map stored in the MMU if this bit is set, even if the map had been previously written to the MMU. The SSM then clears the bit flag, to indicate the map in the MMU is now up to date.

11.5.2 F$CCtl System Call

The higher members of the 68000 family have on-chip memory caches. The 68020 has an instruction cache, while the 68030 and 68040 have separate instruction and data caches. In addition, some processor boards have off-chip caches. In order to be able to support such boards, Microware has not included control of the caches in the kernel. Instead, the caches are controlled by the **syscache** kernel customization module, which installs the **F$CCtl** system call (the kernel's default handler for this system call does nothing, and returns no error). The system call allows the instruction and data caches to be separately enabled, disabled, and flushed (any dirty data is written to main memory, and the current cache contents are forgotten). The kernel uses this system call, for example to disable the data caches during I/O calls.

The parameter passed to the system call is not an image of the processor's **cacr** (cache control) register. Instead, it is a long word of six bit flags, each requesting the instruction or data caches to be enabled, disabled, or flushed. If an undefined bit is set, or a call is made from user state requesting action other than flushing one or both cache sets and the caller is not a super user, a "parameter" error (**E$Param**) is returned. It is not an error to request an action that is not supported by the hardware (for example, enabling the data cache on a 68020).

This system call supports nested requests to disable the instruction or data caches, which are enabled on coldstart. If a request is made to disable the instruction or data caches, the **D_DisInst** or **D_DisData** field respectively of the System Globals is incremented, and the appropriate caches are disabled. If a request is made to enable a set of caches, the appropriate field of the System Globals is decremented (unless it is already zero). If it is now zero, the corresponding caches are enabled, otherwise they are left disabled (and the flag bit in the parameter is cleared). If the parameter contains flags requesting that a cache set be both enabled and disabled, the request to enable the cache takes precedence, and the request to disable the cache is ignored. If the parameter has no bits set this is taken to be a request to flush all the caches.

Note that the current state of the cache control is not maintained separately for each process. Therefore if a process disables caching it does so for all processes. The most recent flag settings are saved in the **D_CachMode** field of the System Globals. The parameter passed to the system call is:

Register	Size	Description
d0	l	Pattern of bit flags.

The bit flags are defined as assembly language symbols in the file 'DEFS/process.a'. They are:

Bit Number	Name	Description
0	b_endata	Enable the data cache(s).
1	b_disdata	Disable the data cache(s).
2	b_fldata	Flush the data cache(s).
4	b_eninst	Enable the instruction cache(s).
5	b_disinst	Disable the instruction cache(s).
6	b_flinst	Flush the instruction cache(s).

The example below shows an assembly language function to make this system call, and a C call to it requesting that the data caches be disabled:

```
void dis_data_cache()           /* disable the data cache(s) */
{
    cache_ctl(0x02);            /* set bit 1 to disable data cache(s)
*/
}

#asm
cache_ctl:  os9     F$CCtl      the parameter is already in d0.l
            rts
#endasm
```

11.5.3 F$ChkMem System Call

This system call checks whether a process has permission to access a memory area, by searching the SSM memory map of the process. The parameters to the call are:

Register	Size	Description
d0	l	Size of the memory area.
d1	b	Access permissions requested - read, write, execute (same format as disk file modes byte).
a2	l	Address of the memory area.

If the SSM is not used (and so has not installed a handler for this system call), the kernel's default handler simply reads the first byte of the memory area. This will generate a bus error if the memory is not accessible, or does not exist.

The kernel uses this system call whenever a system call made from user state passes a pointer to a memory area for the system call to read or write - for example, an I/O "read" (**I$Read**) request.

11.5.4 F$DatMod System Call

This system call creates a module in memory. Prior to OS-9 version 2.3 only a data module could be created, and the use of coloured memory was not possible. From OS-9 version 2.3 onwards an extension to the system call allows the module type and language to be specified explicitly, and a memory colour to be specified. These two extra parameters are used only if bit 15 of the **d2** register (module permissions) is set, otherwise the system call assumes a type of "data", a language code of zero, and a colour of zero (general system memory). The parameters passed to the call are:

Register	Size	Description
d0	l	Size of the body of the module desired (excluding the header, CRC, and name string).
d1	w	Module attributes and revision number.
d2	w	Module access permissions. Also, if bit 15 is clear, registers **d3** and **d4** are ignored.
d3	w	Module type and language.
d4	b	Memory colour to use (zero means general system memory).
a0	l	Address of the name of the module to create.

The kernel allocates memory for the module, builds the module header, clears out the module body, sets the module CRC, and installs the module in the module directory. The "execution offset" field (**M$Exec**) of the module header gives the offset from the start of the module header to the module body - the memory for use by the program. The values returned are:

Register	Size	Description
d0	w	Module type and language.
d1	w	Module attributes and revision number.
a0	l	Caller's register updated past name string.
a1	l	Address of module body ("execution entry").
a2	l	Address of module header.

11.5.5 F$DelTsk System Call

The kernel makes this system call when terminating a process. It indicates to the SSM that the task number (if any) which is allocated to the process can be released for use by another process. Or, if the MMU can only store one map, that the SSM should forget that the MMU contains the map for this process (in case the process descriptor memory is reused for another process). The SSM also de-allocates any remaining memory that it had allocated for the management of this process's memory map.

11.5.6 F$FModul System Call

The **F$Link** system call uses this function to locate a module in the module directory, given the module name, type, and language. The parameters to the call are:

Register	Size	Description
d0	w	Module type (high byte) and language (low byte) (or zero to ignore type or language).
a0	l	Address of the module name string.

This system calls searches the module directory for the desired modules, and returns:

Register	Size	Description
d0	w	Actual module type and language.
d1	w	Module attributes (high byte) and revision number (low byte).
a0	l	Updated past the module name.
a2	l	Address of the module directory entry.

11.5.7 F$GBlkMp System Call

This system call returns information about the free memory areas on the system. The parameters passed to the call are:

Register	Size	Description
d0	l	Memory areas whose start address is below this value are not included in the returned segment list (but are included in the segment count and free memory total).
d1	l	Size of the caller's buffer to contain the returned segment list (in bytes).
a0	l	Address of the caller's buffer.

The system call routine scans the free memory lists, counting the number of separate segments of memory that are free in the system, and totalling their size. For each such segment whose start address is above the specified minimum passed in the **d0** register, the start address and size are written to the buffer (as long words, in that order), until the buffer is exhausted. If the buffer is not exhausted when all segments have been scanned, the next entry in the buffer is cleared to zeros (two long words).

If a segment lies in a memory area that is not designated as "user" memory in the coloured memory list in the **init** configuration module, the segment is not included in the totals or in the table. The values returned from the call are:

Register	Size	Description
d0	l	System minimum allocatable block size, copied from the **D_BlkSiz** field of the System Globals.
d1	l	Number of separate free segments of memory found.
d2	l	Total amount of RAM found at startup, copied from the **D_TotRAM** field of the Systems Globals.
d3	l	Total of free user memory at present.

11.5.8 F$GProcP System Call

This system call returns the address of a process descriptor, given the process ID. The parameters passed to the call are:

Register	Size	Description
d0	w	The process ID.

The values returned from the call are:

Register	Size	Description
a1	l	Address of the process descriptor.

11.5.9 F$Gregor System Call

This is the complement to the **F$Julian** system call. It converts a Julian date and time to Gregorian format. The parameters to the system call are:

Register	Size	Description
d0	l	Julian time.
d1	l	Julian date.

The system call returns:

Register	Size	Description
d0	l	"Gregorian" time.
d1	l	Gregorian date.

A Julian date is simply the number of days from a reference date. A Julian date of zero corresponds to a Gregorian date of 2nd January, in the year -4712. The Gregorian date of the 1st January, 1900, corresponds to a Julian date of 2415020. A Julian time is simply the number of seconds since midnight. While Gregorian dates and times are more natural to humans, the

Julian equivalents are easier to manipulate numerically. Therefore OS-9 provides system calls to translate between the two representations.

Under OS-9 Gregorian dates are held within a single long word, with bits 16:31 containing the year (since 0 AD - for example, 1992), bits 8:15 containing the month, and bits 0:7 containing the day of the month. This is often represented as **YYYYMMDD**. Similarly, "Gregorian" times are held in a single long word, with bits 16:23 containing the hour (24 hour clock), bits 8:15 containing the minute, and bits 0:7 containing the second. This is often represented as **00HHMMSS**.

11.5.10 F$GSPUMp System Call

This system call returns a generalized representation of the memory map of a process. The SSM converts its memory map for the process into a standard table form in the caller's buffer. The table is an array of word values, one for each memory block in the address space of the processor, where the block size is the System Minimum Allocatable Block Size (**D_BlkSiz** in the System Globals) - the block size supported by the MMU. A typical block size is 4k bytes. For a processor with a 32 bit address bus (as the 68000 family has) - a 4 Gigabyte address space - this would give a table 2M bytes in size!

To avoid the need for such a large buffer, Microware have taken into account that in reality most systems have all their user memory low down in the address space. The System Globals field **D_AddrLim** contains the address of the highest memory location (both RAM and ROM). The caller can use this field, and the **D_BlkSiz** field, to determine the size of table to allocate. In any case, the SSM will not return information for address space blocks above this limit.

Many systems have the RAM low down in the address space, and the ROM high up in the address space. This can result in a very large desired table size. The calling program may decide that it does not need the mapping information for the ROM areas. The memory list in the **init** module can be searched to determine the actual extent of the RAM space on the system.

The table is therefore a representation of the address space of the processor, or a part of it starting at address zero. For example, if the system has 4M bytes of RAM starting at address zero, and the System Minimum Allocatable Block Size is 4k bytes, the table requires 1024 word entries (2048 bytes). For each block of 4 kbytes, starting at address zero, the SSM will build a table entry with the following format:

High byte: Access permissions

Bit	Permission if set
0	Read
1	Write
2	Execute

Low byte: Use count.

If the block is not in the memory map of the designated process, the entry is set to zero. The SSM also returns the System Minimum Allocatable Block Size (a copy of the value in **D_BlkSiz**). The parameters to the system call are:

Register	Size	Description
d0	w	Process ID of the process whose memory map is desired.
d2	l	Size of the buffer for the table (in bytes).
a0	l	Address of the buffer for the table.

The system call returns the following values:

Register	Size	Description
d0	l	System Minimum Allocatable Block Size.
d2	l	Size of the table (in bytes) - equal to the buffer size, unless limited by **D_AddrLim**.

11.5.11 F$IODel System Call

This system call checks that a module is not in use by any device table entry. The kernel makes this call internally whenever a file manager, device driver, or device descriptor module is unlinked, reducing the link count to zero. If the module is is in use by a device table entry, the kernel leaves the module's link count at one, and returns an error number 209 (**E$ModBsy**). The parameters to the call are:

Register	Size	Description
a0	l	Address of the module to check.

11.5.12 F$Load System Call

This system call reads a file containing one or modules, allocating memory for the modules, and installing them as a module group in the module

directory. If the "read" bit is set in the file access modes parameter, and the "execute" bit is not set, the file is opened relative to the process's current data directory. Otherwise it is opened relative to the process's current execution directory. Prior to OS-9 version 2.3 it was not possible to specify the colour of the memory to load the modules into. From OS-9 version 2.3 onwards an additional parameter giving an explicit memory colour is taken if bit 7 of the **d0** register (the file access modes) is set, otherwise general system memory is used. The parameters passed to the call are:

Register	Size	Description
d0	b	File access modes. Also, if bit 7 is clear, register **d1** is ignored.
d1	b	Memory colour to use (zero means general system memory).
a0	l	Address of pathlist of file to load from.

The values returned are:

Register	Size	Description
d0	w	Module type and language.
d1	w	Module attributes and revision number.
a0	l	Caller's register updated past pathlist string.
a1	l	Address of program start ("execution entry").
a2	l	Address of module header.

If the file contains more than one module, the returned values are for the first module in the file. The link count of the first (or only) module is set to one. The link counts of any other modules in the file are set to zero. See the chapter on OS-9 Memory, Modules, and Processes for a description of module groups.

11.5.13 F$Permit System Call

If the SSM is in use, a process cannot normally access memory other than memory allocated to it, or modules it has linked to. An attempted illegal access will fail (a "write" will not affect the destination memory), and a bus error exception will be generated. However, in some applications it is necessary for a program to access other areas of memory. This system call allows a process to gain access to any memory area. The **F$Permit** system call is also used internally by the kernel whenever it wishes to add a memory area to a process's memory map - for example, when a process allocates memory, or links to a module.

As with other SSM system calls, the functionality of this call depends on the implementation within the SSM. The description here is of the functionality of a Microware SSM for the memory management unit in the 68030 processor. This system call adds a memory area to the memory map of a process, permitting the process to access that memory. The parameters to the call are:

Register	Size	Description
d0	l	Size of the memory area.
d1	b	Access mode requested (read, write, execute) in the same format as path modes.
a2	l	Address of the memory area.

If the call is made from user state, an error is returned unless the call is made by a member of the super user group (group zero). If the process was forked by the **F$DFork** system call, as a debugged process, the memory area is also added to the parent's memory map. This is recursive, so a debugged process can fork a debugged process, and so on.

The memory address is rounded down to the nearest whole block of the size supported by the MMU (the System Minimum Allocatable Block Size). The size is rounded up to a whole number of blocks, to include the start and end addresses of the area requested. This ensures that the whole of the desired memory area is accessible to the process, without the process needing to know the system minimum allocatable block size.

If the process is the System Process (the process ID is one), the system call does nothing.

Read and execute permissions are always granted (the 68030 MMU does not distinguish between them). Write permission is only enabled if requested in the access modes.

The SSM must take account of the possibility that **F$Permit** will be called more than once for the same process and memory area. For example, a process might link to the same module several times. Therefore the SSM searches the process's memory map to see if the block is already in the process's memory map. If so, the SSM just increments a use counter that it keeps for each block in a process's memory map. Otherwise it adds the block to the process's memory map, and sets the use counter for that block to the initial value of one.

This feature is necessary so that when a request is made to unmap the block - for example, when a module is unlinked - it is only actually removed from the process's map when the block is no longer required for any reason. (See the description of **F$Protect** below). Note that the precision of the use count may be limited - for example, the SSMs for the 68030 and 68040 maintain an 8 bit use counter for each block. Therefore a possibility of error exists if a block is "mapped to" more than 255 times by the same process (for example, if a module is linked to more than 255 times by the same process). The SSM limits the counter to 255, so after 255 "unlinks" the module will be removed from the map of the process, even though the process believes it is still linked to the module.

If the MMU has internal memory maps, the SSM does not update the MMU during this system call. Rather, it updates the map in memory that will be copied to the MMU when the process is next about to execute in user state (see the description of the **F$AllTsk** system call above). While in system state the MMU is configured to remove all protections - operating system functions (and system state programs and trap handlers) have unrestricted access to the full memory map of the processor.

If the SSM is not used (and so has not installed a handler for this system call), the kernel's default handler simply reads the first byte of the memory area. This will generate a bus error if the memory is not accessible, or does not exist, causing the calling process to be aborted (unless it has installed a bus error handler - see the **F$STrap** system call, and the chapter on Exception Handling).

11.5.14 F$Protect System Call

This system call is the complement to **F$Permit**. It removes a memory area from the memory map of a process, denying the process access to that memory. The parameters to the call are:

Register	Size	Description
d0	l	Size of the memory area.
a2	l	Address of the memory area.

If the call is made from user state, an error is returned unless the call is made by a member of the super user group (group zero). If the process was forked by the **F$DFork** system call, as a debugged process, the memory area is also removed from the parent's memory map. This is recursive, complementing the nested **F$DFork** calls supported by **F$Permit**.

As with **F$Permit**, the memory address is rounded down to the nearest whole block of the size supported by the MMU (the System Minimum Allocatable Block Size). The size is rounded up to a whole number of blocks, to include the start and end addresses of the area requested.

If the process is the System Process (the process ID is one), the system call does nothing.

The system call decrements the use count in the process's memory map for each block in the memory area (see the description of **F$Permit** above). If the use count for a block reaches zero, it is removed from the process's memory map.

If the SSM is not used (and so has not installed a handler for this system call), the kernel's default handler simply reads the first byte of the memory area. This will generate a bus error if the memory is not accessible, or does not exist.

The kernel uses this system call whenever a memory area is de-allocated by a process, or the process unlinks from a module.

11.5.15 F$SysDbg System Call

This system call causes the kernel to call the "system debugger", if one is present. The kernel makes a subroutine call to the routine whose address is in the **D_SysDbg** field of the System Globals. The kernel only checks that the caller is the super-super user (user zero of group zero). It does not set up any registers for the "system debugger" (although the **a6** register contains the address of the System Globals). From OS-9 version 2.3 onwards, the kernel makes the **F$CCtl** system call to flush and disable the data and instruction caches before calling the debugger, and to flush and enable the caches on return from the debugger.

The **D_SysDbg** field of the System Globals is initialized during the kernel's coldstart to the "boot entry point" address passed from the boot program, plus 16. In the boot program, this should point to a branch instruction to the ROM-based debugger. Therefore executing the **F$SysDbg** system call normally invokes the ROM-based debugger. This halts the normal operation of the system. The ROM-based debugger will continue normal system operation in response to the "go" (g[CR]) command.

11.5.16 F$SysID System Call

The kernel header contains a licensee number, a serial number, the processor type this kernel supports, and (in an encrypted form prior to OS-9 version 2.3) a version description string and a copyright string. The body of the kernel also contains an author names string in an encrypted form. This system call returns these items (with the strings decrypted as necessary), together with the processor type in use (determined dynamically by the boot program).

The processor type numbers are the Motorola part number: 68000, 68010, 68020, 68030, 68040, 68070 and so on. For example, a 68010 processor running the 68000 version of the OS-9 kernel would give a kernel processor type of 68000, and the processor type in use as 68010.

The strings returned are null terminated, and will not be longer than 80 characters, including the null. The parameters to the system call are:

Register	Size	Description
a0	1	Address of the buffer for the version string.
a1	1	Address of the buffer for the copyright string.
a2	1	Address of the buffer for the author names string.

If a buffer address is passed as zero, the system call handler does not attempt to copy that string. Apart from copying the strings to the buffers, the system call returns:

Register	Size	Description
d0	1	OS-9 licensee number.
d1	1	Serial number of this copy of OS-9.
d2	1	Processor type in use.
d3	1	Kernel processor type.
d4-d7	1	Zero.

The licensee number and serial number are one by default, but may be used by Microware or the licensee (the manufacturer of the computer system) to identify the licensee and the individual copy of OS-9, as a piracy protection measure.