CHAPTER 3

OS-9 MODULES, MEMORY, AND PROCESSES

3.1 THE OS-9 MEMORY MODULE

The OS-9 memory module concept is at the heart of many of the innovative features that make OS-9 applicable to such a wide range of applications. It permits the operating system to keep track of programs, operating system components, and common data areas in memory. Irrespective of the absolute location of these items in memory, programs can locate them by passing a name to the operating system, similar in some ways to accessing a file on a disk drive.

An OS-9 module is a program, data structure, operating system component, or any string of bytes, with an identifying header stuck on the front and a Cyclical Redundancy Check (CRC) tagged on the end.

The header contains information about the module, to prevent its unauthorized or incorrect use, and facilitate automatic mechanisms such as finding all modules in ROM at startup, initialization of the data space of a program, and protection against data corruption by module CRC checking. Very importantly, the header also contains an offset to the module name string.

A module is known by its name. All modules currently in memory must have different names (unless of a different type or language - see below), although modules with the same name may be contained in files (for example, on disk).

The addresses of all modules in memory are held in the "module directory", which is a table built and maintained in memory by the operating system. Each entry in the module directory contains the address of the module, the current number of "links" (uses) to the module, a group identifier (explained

later), and a module header parity check value, to guard against corruption of the module header.

When a program or operating system function wants to locate a module in memory, it makes an operating system "link to module" call (**F$Link**), passing the name of the module. The kernel looks at each module directory entry in turn. The module address points to the module header, from which the kernel finds the name of the module, and compares this with the name passed to it. This is repeated until it finds the module requested.

[The term "link" causes some confusion, as it has many meanings in software terminology. In the OS-9 environment it has two principal meanings:

a) To get the address of a module in memory, by using the operating system **F$Link** system call.

b) To build a program (or other module) from one or more **Relocatable Object Files** (ROFs), using the "linker" utility.

There is no connection at all between these two functions. The OS-9 linker utility is called **l68**.]

Modules may be present in ROM at startup, or be loaded from disk (or other I/O device) at any time. When a module is to be loaded, the kernel allocates memory equal to the size of the file, and reads the file into that memory. It then checks the memory for valid modules, adding them to the module directory. Some operating system calls cause a module to be implicitly loaded from disk - for example, if a program fork is requested, and the program module is not already in memory.

Modules explicitly loaded from disk (such as by the **load** utility) are retained in memory even while not being used. For example, commonly used utilities can be loaded, using up memory, but saving on disk accesses. When the module is no longer required it can be removed from the module directory and the memory released, by "unlinking" the module. This is the function of the **unlink** utility. The module header may have the "sticky" (or "ghost") flag set. In this case a module that has been *implicitly* loaded (such as by a "fork" request) will stay in memory after it has been used, until its memory is needed for something else. The Microware-supplied utilities are sticky modules.

Programs must be in modules. A module may be loaded by the operating system anywhere in free memory. Therefore programs (in general) must be position-independent. Their data space may be anywhere in free memory, and is separate from the program space. This allows program modules to be directly ROMmable and re-entrant (as explained earlier).

The operating system itself is broken up into several modules, facilitating customization and expansion. The module system means that no explicit "system build" is required. All modules in ROM and in the boot file are found on startup and installed in the module directory.

3.2 A PROGRAM MODULE

The module header and CRC are automatically generated by the linker when a program (or other module) is assembled or compiled, so there is no added complication to creating programs. The module header consists first of a universal section - the same structure in all modules - followed by a parity check word. After this comes the second part of the header, which varies depending on the purpose of the module, as specified by the module "type" code in the universal part of the header.

As an example, the layout of a program module is shown in figure 5 on the next page. As a module can be loaded anywhere in memory, items within the module cannot be referred to by their absolute memory address. Instead, they are identified by their offset from the beginning of the module header (as if the module were loaded at address 0). In the following description (as throughout this book), a byte is 8 bits, a word is 16 bits, and a long word (or just "long") is 32 bits. A prefix of '$' indicates a hexadecimal (base 16) number. The offsets from the start of the module header are given in hexadecimal. The C symbolic names are taken from the file 'DEFS/module.h'. There is also an assembly language file 'DEFS/module.a' - the structure item names are similar, though not the same.

3.2.1 Sync Word

The value $4AFC is an illegal instruction in the 68000 family instruction set, and so occurs very rarely in memory. Starting the module header with this "magic number" speeds up the kernel's search for modules in ROM at coldstart. It checks each word in ROM - only if the word is $4AFC does the kernel attempt to check the module header parity and Cyclical Redundancy Check. The sync word is also useful to the user when looking through memory.

Offset	Size	C name	HEADER
$0000	word	_msync	Sync word $4AFC
$0002	word	_msysrev	System revision ID
$0004	long	_msize	Module size
$0008	long	_mowner	User/group of creator
$000C	long	_mname	Offset to module name string
$0010	word	_maccess	Module access permissions
$0012	word	_mtylan	Type and language
$0014	word	_mattrev	Module attributes and revision
$0016	word	_medit	Edition number
$0018	long	_musage	Offset to usage comments string
$001C	long	_msymbol	Offset to symbol table
$0020	word	_mident	Ident code
$0022		_mspare	12 bytes reserved
$002E	word	_mparity	Header parity
-----	----		EXTENDED HEADER
$0030	long	_mexec	Offset to program entry point
$0034	long	_mexcpt	Offset to default trap entry point
$0038	long	_mdata	Minimum program data space
$003C	long	_mstack	Minimum program stack size
$0040	long	_midata	Offset to data initialization table
$0044	long	_midref	Offset to data pointers initialization table
-----	----		----------
$0048			PROGRAM BODY
-----	----		----------
	long		CRC

• **Figure 5 – Module header structure.**

3.2.2 System Revision ID

A number indicating which version of the module header follows. So far the module header structure has not changed, but this field allows for backward compatibility if the header structure is changed in future versions of OS-9.

The current value in this field is $0001.

3.2.3 Module Size

The size of the whole module, including the header and CRC.

3.2.4 User and Group

This field gives the group number (in the high, or first word of the long word) and user number of the user who created the module. For example, if the user creating the module were user 4 of group 2, this field would contain $00020004. This allows modules to be protected against use by other users, if desired, by clearing the appropriate flags in the access permissions field.

If a use of a module (link, fork, use in an I/O sub-system) is attempted by the same user (same user ID and group number) as the module creator, the desired modes (read and execute in the **F$Link** system call, for example) are checked against the private field of the module access permissions. The same check is made if the access is made by a member of the super user group (group zero). If the user has the same group number as the module creator but not the same user ID, the desired modes are checked against the group field of the access permissions. Otherwise, the desired modes are checked against the public ("world") field of the access permissions. If the appropriate permissions flags are not set, the attempted system call fails with a "no permission" (**E_PERMIT**) error.

The kernel also uses the user/group field as a privilege mechanism. Only modules created by a user of group zero (the super-user group) can make certain system calls. The super-user group can compile programs to give controlled access to certain resources by other users, who can run the programs.

3.2.5 Offset to Module Name

The linker adds the desired module name string (terminated with a null character, binary zero) to the body of the module, and sets this field with the offset (from the start of the header) to the name string. The module directory entry contains the address of the module (start of the header). The kernel reads the offset to the name string from the header, adds it to the address of the module, and so can compare the name of the module to the name provided by a program that wishes to locate ("link to") the module.

3.2.6 Access Permissions

Only the low twelve bits of this word are used, as three groups of four bits:

Bit	Permission
0	Private read (creator only)
1	Private write
2	Private execute
3	reserved
4	Group read (members of creator's group only)
5	Group write
6	Group execute
7	reserved
8	Public read (any user)
9	Public write
10	Public execute
11	reserved

On a system *without* the System Security Module software (and memory management hardware) the flags have the following effect when set:

read	allows load, link, and unlink
execute	allows load, link, unlink, and fork
write	no function

On a system *with* the System Security Module software and memory management hardware, the flags have the following effect when set:

read	allows load, link, and unlink
execute	allows load, link, unlink, and fork
write	allows writing to the module in memory once linked to

A violation of any of these permissions when attempting to gain access to the module (for example, by linking to it) gives an error number 164 (**E_PERMIT**).

3.2.7 Type and Language

The "type" code is in the high (first) byte, while the "language" code is in the low byte. The type code indicates the intended usage of the module. Microware have reserved codes 0 to 15, of which the following codes are currently used:

Code	Module type
1	program
2	subroutine
4	data module
11	trap handler
12	operating system component (other than I/O)
13	file manager
14	device driver
15	device descriptor

The kernel will check the type code against an attempted usage. For example, trying to fork a module that is not type 1 gives an error number 234 (**E_NEMOD**). If a program specifies a type code of zero when trying to link to a module, the kernel will allow the link whatever the actual module type. Otherwise, the type code must match.

Note that it may not be advisable at present for users to make use of the undefined type codes (16 to 255). This is because the "unlink" system call (**F$UnLink**) assumes that any module with a type code of 13 or higher is part of the I/O system, causing a search of the device table to see if the module is still in use. However, this should not cause a problem, as the module will (presumably) not be found in the device table.

The language code indicates the encoding of the module body. Microware have reserved codes 0 to 15, of which the following codes are currently used:

Code	Language
1	object (machine code) executable
2	compiled Basic intermediate code

Other codes are defined, but have no use at present:

Code	Language
3	compiled Pascal intermediate code
4	compiled C intermediate code
5	compiled Cobol intermediate code
6	compiled Fortran intermediate code

The kernel will check the language code against an attempted usage. For example, trying to fork a module that is not type 1 gives an error number 234 (**E_NEMOD**). If a program specifies a language code of zero when trying to link to a module, the kernel will allow the link whatever the actual module language. Otherwise, the language code must match.

The **shell** uses the language code to automatically fork up an appropriate run-time interpreter for intermediate code programs. For example, if the language code is 2, **shell** will fork the program **runb** to run the Basic compiled intermediate code program.

Note: the kernel will allow multiple modules of the same name in the module directory, provided they have a different type and/or language code. A **link** (or an **unload** - an unlink by name) with a type and language code of zero, will operate on the first module of that name in the module directory.

3.2.8 Attributes and Revision

The module attributes are in the high (first) byte of this field. The revision number is in the low byte.

☐ Module attributes

The attributes are a set of bit flags, indicating special treatment by the kernel:

Bit	Meaning when set
7	module is sharable
6	sticky module
5	supervisor (system) state module

If the module is sharable, the kernel will permit a link count greater than one. This flag is normally set for all modules. If this flag is clear, only one link is permitted at a time. A device descriptor with this bit clear can only

have one path open to it at a time. A program module with this bit clear is not re-entrant - there can only be one incarnation of the program at any one time - which would allow it to be self modifying.

A "sticky" module remains in memory even when its link count has been reduced to zero. A further reduction in its link count (by the **F$UnLink** system call, for example from the **unlink** utility) causes the module to be removed from the module directory, and its memory returned to the free pool. Also, if the operating system receives a memory request that it cannot satisfy from the existing free pool, it will remove sticky modules whose link count is zero from the module directory (in the order of their entry in the table) until enough memory is available. This mechanism allows commonly used modules to remain in memory after use until they are needed again, or until their memory space is needed.

The "supervisor state" flag is set for all operating system components, and for programs (and trap handlers) that are to run in supervisor (rather than user) state. The 68000 family microprocessors have two operating states - supervisor and user. The processor's internal status register has a bit that indicates supervisor state when set, user state when clear. Certain instructions are not permitted in user state. Any operating system call or external interrupt automatically puts the microprocessor into supervisor state, so all operating system components run in supervisor state. This mechanism provides essential protections between programs in a multi-user system, and (with memory management hardware) prevents programs from corrupting the operating system.

For these protections to work, programs run in user state. In certain applications, however, it may be advantageous to run a program (or trap handler) in supervisor state. This requires great care, and a thorough understanding of the effects (described in the chapter on OS-9 System Calls).

Supervisor state is often called system state, because the operating system runs in this state. In this book the term "supervisor state" refers to the physical operating state of the microprocessor, while "system state" refers to the logical state of the software. For example, a different microprocessor might not have a supervisor state, but the computer would still be logically in system state when executing an operating system function or interrupt service routine.

In OS-9 supervisor state and system state are synonymous. The operating system tests the supervisor state bit of the processor's status register to determine if the computer is in system state.

☐ Module revision number

The revision number feature allows modules to be superseded without removing them from memory. It is useful as a means of overriding out-of-date modules in ROM, or in the boot file (the boot file is ROM as far as the operating system is concerned).

When checking a module in ROM on coldstart, or when loading a module into memory, the kernel scans the module directory to see if a module of the same name, language code, and type code is already in the module directory. If so, the kernel compares the revision numbers of the two modules. If the new module has a higher revision number than the module already in the module directory, the kernel overwrites the existing module directory entry with the information about the new module.

If the two modules have the same revision number, the kernel checks that the following conditions are satisfied:

1) The system supports "sticky" modules (the **B_Ghost** bit of the **D_Compat** byte of the System Globals is set).

2) The link count of the old module is zero.

3) The link counts of any other modules in the same module group are zero.

4) The new module is not in the same module group as the old module (this check was omitted in OS-9 version 2.2).

5) The kernel has finished its coldstart (the **D_ID** field of the System Globals contains $4AFC).

If all these conditions are satisfied, the kernel frees the memory of the old module and overwrites the existing module directory entry with the information about the new module. If any of the conditions is not satisfied, or the revision number of the new module is lower than the revision number of the old module, the kernel ignores the new module (and returns the memory

used to read in the file, in the case of a **load**), and returns error 231 - "module already known" (**E_KWNMOD**).

It is very important to note that if the new module's revision number is higher than that of the old module, the kernel makes no checks to ensure that the old module is not in use before it deletes the module directory entry. Also, the kernel does not return the memory of the old module to the free pool (because it assumes the module is in ROM). Therefore this feature must be used with care.

3.2.9 Edition Number

This is a software maintenance edition number for the module. It is set by the programmer when the module is created, and is not used by the kernel. It allows a user to identify the edition of a program or operating system component when asking for technical assistance from the software supplier.

3.2.10 Other Fields

☐ Offset to usage comments string

This field is not used at present.

☐ Offset to symbol table

This field is not used at present.

☐ Ident code

The ident code is intended to be used by the **ident** utility when inspecting a module header to display information about the module. **ident** does not use it at present.

3.2.11 Header Parity

This word is the one's complement of a word-by-word exclusive OR of all the preceding words in the header. Therefore the header parity is correct if a word-by-word exclusive OR of all words in the header up to *and including* the header parity gives a result of $FFFF.

While checking the module header parity prior to installing the module in the module directory, the kernel also calculates a "checksum" over the

module header, and saves this value in the module directory entry for the module. When the kernel receives a request to link to the module, it rechecks this module header checksum, and verifies that the header checksum word now calculated from the header matches the value saved in the module directory. If there is a change, the kernel assumes that the module is corrupt, and returns an error number 236 (**E_BMHP**). This gives some protection against disastrous system errors that might occur if the kernel were to use the entries in the header of a module that has been corrupted after being loaded into memory.

In OS-9 version 2.2 this checksum was simply the low 16 bits of the sum of all the words in the universal module header, including the parity word. In later versions the calculation is slightly more complex - after each word addition the word result is rotated right by a number of bits equal to its own value (modulo 16).

3.2.12 Offset to Program Entry Point

Also known as the "execution offset", this is the offset from the start of the module header to the first instruction to execute in a program. From this the kernel can calculate the absolute address at which to start program execution, by adding this field to the actual address in memory of the module. The first instruction to execute need not be the first instruction in the module - it can be anywhere in the module.

The linker calculates this offset when creating the module, using the entry point offset given in the "root" **psect** in the files used to create the module. The linker can create a program from multiple files, but there must be one and only one file containing a "root" **psect** (described in the chapter on the C Compiler, Assembler, Linker, and Debugger), so no confusion can arise.

The C compiler only produces non-root **psects**. When a C program is linked, the 'cc' executive program adds the file 'LIB/cstart.r' at the front of the list of files to link. 'cstart.r' (created from 'C/SOURCE/cstart.a') is a root **psect**, containing a function to initialize the program and then call the **main()** function of the program.

3.2.13 Offset to Default Trap Entry Point

This offset is produced and used in a similar way to the execution offset. The kernel uses it to calculate the address of the program function to call if the program uses a 68000 **TRAP** instruction for which it has not installed a trap handler module.

3.2.14 Minimum Program Data Space

When creating a module, the linker adds up the static storage (variables) definitions in each of the Relocatable Object Files to calculate the total static storage requirement of the program, and sets this field with that value.

3.2.15 Minimum Program Stack Size

The linker sets this field with an assumed maximum stack usage by the program. As program functions can be recursive (call themselves directly or indirectly), the linker cannot actually calculate a real maximum stack requirement. It uses a default value (3k bytes), which can be overridden by a command line option to the linker.

The kernel adds together the "minimum data space" and the "minimum stack size" fields to calculate the total memory required (initially) by the program. It adds to this the size of the parameter string being passed to the program by the parent process (because the kernel copies the parameter string from the parent's buffer to the top of the child's memory space), and the "additional stack size" parameter passed to the **F$Fork** system call. The result is the total memory the kernel must allocate for the new process. The low part of the memory allocated is used for the static storage, the middle part for the stack, and the top part for the copy of the parameter string.

3.2.16 Offset to Data Initialization Table

The offset from the start of the module header to a table (built by the linker) of data values for use by the kernel when initializing the program's static storage. The C language permits static storage variables to be initialized with constant values at startup. The kernel uses this table to implement this feature.

The table structure is as follows:

Offset	Size	Description
$0000	long	Offset within static storage to start of area to initialize.
$0004	long	Size of area to initialize, in bytes.
$0008	bytes	The initialization data.

The kernel copies the data from the table to the indicated part of the static storage.

The linker separates out all initialized static storage definitions from all uninitialized static storage definitions. It locates the initialized static storage after the uninitialized static storage, so the initialization table is a direct image of the desired initialized variables in memory. Because the linker separates out the static storage definitions in this way, the location of variables in memory may differ from that expected. However, the order of the variables is maintained (but separated into the two areas). This is analogous to splitting a single mixed queue of badgers and foxes into two separate queues.

3.2.17 Offset to Data Pointers Initialization Table

The offset from the start of the module header to two tables (built by the linker) of structures for initializing pointer variables in static storage. The C language permits static storage pointer variables to be initialized with the addresses of other static storage locations, or of program functions.

As the absolute address of neither the program nor the static storage is known when the linker creates the module ("link time"), the linker cannot put the required absolute addresses for these variables into the "data initialization table" described above – it can only put the offset from the start of the module header (for program function pointers) or static storage (for static storage pointers). When the program is forked the kernel (which now knows the address of the program module and of the static storage it has just allocated) must adjust these initialized values in the pointer variables. The kernel locates the pointers that must be adjusted using the information in these two tables.

The first table contains information for initializing pointers to program functions. The kernel must add the address of the start of the module header to the offsets in this table to form the absolute pointer values. The second table contains information for initializing pointers to static storage locations. The kernel must add the address of the start of the static storage to the offsets in this table to form the absolute pointer values.

Each table consists of zero or more lists. Each list has the following format:

Offset	Size	Description
$0000	word	high (most significant) word of the offset to use
$0002	word	number of entries in the list
$0004	words	the list – each word is the low (least significant) word of the offset to use

The table ends with a long word of zero - that is, when looking for the next list, the kernel reads the "high word" and "number of entries" values, stopping when the "number of entries" is zero. The second (static storage pointers) table follows immediately after the terminating long word for the first table.

For each entry in the list, the kernel reads the "low word" value and adds it to the common "high word" value (shifted left 16 bits) for the list, to form a long word offset into the static storage, from which it calculates an absolute address within the static storage. It then adds to the long word (pointer variable) pointed to by that address to either the start address of the module header (first table) or the start address of the static storage (second table).

Because the static storage pointer variables are "initialized static storage", they are included in the "data initialization table" described above. This has already initialized them with the offset (calculated by the linker) from either the start of the module or the start of the static storage to the item they are intended to point to. This second operation of adding the start address of the module header or of the static storage changes the offset into the required absolute address.

3.2.18 Module CRC

OS-9 uses a 24 bit Cyclical Redundancy Check value at the end of the module. The linker adds a byte of zero to the end of the module body before the CRC to ensure that the module is an even number of bytes long - the 68000 family of processors requires that all instruction words be on an even address.

The CRC is generated by the linker and checked by the kernel using the "generate CRC" (**F$CRC**) system call. This call takes an initial accumulator, a byte count, and a pointer to the bytes over which to calculate the CRC. It returns the new accumulator value. Using this call a module's CRC can be generated piece by piece, or all at once. Initially the accumulator must be set to $FFFFFF. When generating a CRC the final value must be one's complemented before adding the three bytes to the module. The linker generates the CRC over the whole of the module, from the first byte of the header to the zero byte before the CRC. When checking a CRC, the three bytes of CRC must be included in the calculation. The result (for a good CRC) is $800FE3.

The kernel does not recheck the CRC of a module after it has been installed in the module directory. Therefore the contents of a module (but *not* the

header - see **Header Parity** above) can be altered. This is clearly necessary for data modules (which are pools of data shared between multiple programs). It is also used by the debugger programs for placing "hard" breakpoints in the modules being debugged.

The CRC calculation is not trivial - it requires a significant processor effort (although the time taken to perform the CRC calculation is not noticeable when loading a program, especially with the higher performance members of the 68000 family). Therefore highly time-critical applications should load all the modules they need before starting the application proper. This is good practice in any case, to reduce the possibility of disk usage affecting a time-critical application.

3.3 MODULES IN FILES

Modules may be contained in disk files, just like any other data. One or more modules may be contained in the same file, merged together sequentially in the file. A module - especially a program module - is normally held in a file of the same name as the module, to avoid confusion, but this is not a requirement. To the disk file manager, nothing distinguishes a file containing a module from any other file.

The **load** utility uses the "load" system call (**F$Load**), which allocates an area of memory equal to the size of the file, reads the file into memory, checks the modules in the file, and installs them in the module directory (using the **F$ValMod** system call). The module directory entry link count for the first (or only) module in the file is set to one (the others are set to zero).

The **unlink** utility uses the "unlink" system call (**F$UnLink**), which decrements the link count of a module. If this reduces the link count to zero, the module is removed from the module directory and its memory is returned to the free pool (unless it is a "sticky" module, in which case the module is removed from the module directory if its link count is reduced to -1).

The operating system "fork" call (**F$Fork**) attempts to find a module of the given name in the module directory. Failing that, it loads a file of the same name from the execution directory, and forks the first module in the file (*which could have a different name*). Note: if a module of the right name exists in the module directory, but it is not type "program" or language "object code", the kernel rejects the "fork" request with the error number 234 (**E_NEMOD**), rather than trying to load a file of the given name.

Exiting from a program causes the program module to be unlinked by the kernel. Therefore if it was implicitly loaded by a "fork" its link count is reduced to zero, and it is removed from the module directory (unless it is a sticky module).

3.3.1 Module Groups

OS-9 allocates memory in multiples of a minimum allocation unit (at present 16 bytes). This is because the areas of free memory are connected together in a linked list, with each free memory area having a controlling structure at the start of the memory area. This structure needs a certain amount of memory, so an area of memory smaller than this could not be freed (de-allocated) - it could not be returned to the free memory list.

However, a file containing more than one module could contain modules whose length is not an integral multiple of this minimum block size, so that when the modules are loaded together in memory there may be memory blocks that contain the end of one module and the start of the next. Therefore, if the memory used by each such module could be freed separately, the free memory areas would no longer be integral multiples of the minimum block size, which cannot be permitted. To solve this problem, Microware devised the concept of the "module group".

Multiple modules loaded from one file constitute a "module group". The address of the first module (and therefore of the memory area allocated to the group) is used as a unique "group identifier" for the group. Each module in the group has this identifier in its module directory entry. All modules in the group remain in the module directory (even if they individually have a link count of zero) until the link count of all of them goes to zero.

That is, when - as a result of an "unlink" - the link count of a module goes to zero (or is already zero, and so would go to -1, for a sticky module), the kernel checks all entries in the module directory to see if there are any other members of this module group whose link count is not zero. If so, it does not remove the module from the module directory. Otherwise, it removes all modules in the group from the module directory, and frees the memory for the whole group.

Note that the kernel does not check whether the other modules in the group are sticky modules. Also, if a module (including a sticky module) is unlinked when its link count is already zero, but another module in the group has a non-zero link count, the link count of the first module remains zero.

When a file of modules is loaded, the kernel sets the link count of the first module to one, and the others to zero. Therefore if no use is made of the modules (so their link counts are not increased), unlinking the first module will remove all modules in the group from the module directory.

Modules in ROM and the boot file form module groups of their own (a group for each separate ROM area in the memory lists), with a group identifier equal to the address of the ROM area containing the modules. During its coldstart the kernel scans the ROM (including the boot file in RAM, which the boot program pretends is ROM to the kernel) looking for modules, and puts them in the module directory. It sets the link count of the first module in each group to one. The kernel then links to itself, so setting its link count to one. This holds each group in the module directory. If each module in a group were unlinked so its link count reached zero, all the modules in the group would be removed from the module directory (as for any group), with potentially fatal consequences.

There is very little protection on module unlinking. Unlinking a module that is in use can have fatal consequences. The kernel does not permit the link count of an I/O module (file manager, device driver, or device descriptor) to be reduced to zero if the module is in use by any entry in the device table. Also, OS-9 version 2.4.3 (released in 1992) does not permit the link count of a module to be reduced to zero if it is the primary program module of any existing process, or an installed trap handler module of any existing process.

3.4 OS-9 MEMORY

The kernel provides dynamic allocation of memory. That is, memory is allocated as needed, and when it is no longer needed it is returned to a free pool.

OS-9 does not use memory management hardware to translate physical memory addresses to logical addresses as seen by a program. The program "sees" the memory at its actual physical address. Therefore a request for memory cannot know where the memory will be located. This means that all system memory usage must be register indirect - that is, relative to a processor internal register that has been loaded with the base address of the allocated memory area.

The same simple mechanism is used for allocating operating system memory and program memory, irrespective of whether the request is made from a program or from an operating system component.

A program is initially allocated memory for its static storage variables and stack by the kernel when the program is forked. In addition, the program may dynamically allocate and release additional areas of memory of any size (rounded up to the minimum allocatable block size), up to a maximum of 32 areas at any one time (including the process's static storage). This limit is imposed because the kernel must keep track of the memory areas owned by the program, so that the program memory can be automatically returned to the free pool on program exit (in case the program is abnormally terminated, or does not clean up before exiting).

If a newly allocated memory area is contiguous with an area already allocated to the process, the two areas are merged - only one entry in the table is used. If the newly allocated memory area fills a hole between two areas already allocated to the process, the three areas are merged into one table entry. This approach makes the maximum use of the 32 entries available in the table.

Because a memory management unit is not used to translate addresses, the kernel cannot combine separate physical memory areas to make a single logical memory area. Therefore problems due to memory fragmentation can occur, but in practice they are very rare, unless the application is very tight on memory, or is allocating very large blocks of memory.

The **mfree** utility used with the '-e' option displays the list of currently free memory areas.

3.4.1 Coloured Memory

Note: "coloured memory" and the "memory list" in the **init** configuration module were added to OS-9 in OS-9 version 2.3. Earlier versions of the operating system do not have these features.

The coloured memory concept was devised to solve two problems:

a) It is desirable to be able to control the order of allocation of different memory areas, as some memory areas may be slower in operation than others - that is, memory areas need to have a **priority** number.

b) Some areas of memory may have special properties (for example, a memory area that is accessible to another processor on the same bus), so it is desirable to be able to specifically reference these different types of memory when allocating memory - that is, memory areas need to have a **type** number.

Microware call the different memory types "colours" - hence the term "coloured memory". Giving a special type of memory a unique colour number allows programs to allocate memory of that type without needing to know the absolute address of the memory. It also allows multiple programs to allocate separate areas of the special memory. Furthermore, if the system has memory management hardware which OS-9 is using for inter-task memory protection, a program cannot access memory which has not been allocated to it - coloured memory is a convenient way for a program to gain access to special memory in a way that is guaranteed to be portable to other OS-9 systems. Examples of memory with special properties are:

- Graphics display memory.
- Battery-backed memory.
- Inter-processor communication memory.

The need for prioritizing the allocation of memory areas is quite common in bus-based computers. The memory on the processor board is usually much faster (for the processor to access) than memory on a separate memory board accessed over the bus.

The kernel builds its table of memory areas from the list of possible memory areas in the **init** configuration data module. During coldstart the kernel reads this list, and tests each area in the list, to see how much memory actually exists in that area. Each entry in the list specifies a start address for an area, an end address, a colour number, a priority, and attribute flags such as "read only", for ROM areas that are to be checked for modules on startup, and "user" for memory that can be allocated to user-state programs.

This allows the user to specify the colour, priority, and attributes of each memory area that may be in the system, and to "hide" memory from OS-9 (by not including it in the list). There are two memory allocation system calls - general (or "uncoloured") and coloured. The uncoloured memory allocation system call is **F$SRqMem**, called by the C library function **_srqmem()**. The coloured memory allocation system call is **F$SRqCMem**, called by the C library function **_srqcmem()**. In addition, the system calls to load modules from a file (**F$Load**, C functions **modload()** and **modcload()**), and to create

a module in memory (**F$DatMod**, C functions **_mkdata_module()** and **make_module()**), provide an extended format that specifies the memory colour to use (see the chapter on the OS-9 System Calls).

If there is more than one memory area of the same colour, memory with a high priority will be allocated before memory with a low priority. The uncoloured memory allocation system call allocates memory in priority order irrespective of colour, and will not allocate memory with a priority of zero. Such memory can only be allocated by a coloured memory allocation request, and so is protected from general system usage.

3.4.2 Memory Allocation

OS-9 uses a simple but effective algorithm for the allocation and de-allocation (freeing) of memory. The kernel maintains a separate linked list of free memory areas for each priority value of each colour of memory. Each free memory area has at its start the following structure:

Offset	Size	Description
$0000	long	Address of next area in linked list.
$0004	long	Address of previous area in linked list.
$0008	long	Size of this area (in bytes).

To keep track of these linked lists, the kernel maintains a table - the memory colour node table - in which each entry is a structure, describing each memory area found at startup. The structure gives the memory area start and end addresses, the memory area colour number, allocation priority and attributes, and the total size of the free memory areas within this memory area. It also gives the addresses of the first and last free memory areas in this memory area.

The memory colour node structures in the table are linked together as a doubly linked list, ordered by allocation priority (highest priority area is first in the list).

Offset	Size	Description
$000	long	Start address of memory area.
$004	long	End address plus one of memory area.
$008	long	Address of next (lower priority) memory colour node in list.
$00C	long	Address of previous (higher priority) memory colour node in list.

Offset	Size	Description
$010	long	Address of first free memory area within this memory area.
$014	long	Address of last free memory area within this memory area.
$018	long	Reserved.
$01C	long	Start address of memory area as seen by alternate bus master (local start address plus translation offset given in memory list in **init** module).
$020	long	Sum of sizes of free memory areas in this memory area.
$024	word	Attributes of this memory area (as given in the memory list in the **init** module).
$026	word	Colour number.
$028	word	Priority.

So, to allocate memory of a given colour the kernel walks through the linked list of colour node structures until it finds one of the correct colour that has sufficient free space. From the structure it takes the addresses of the first and last free memory areas within this memory area. It then walks through that linked list of free memory areas, checking the size of each entry until it finds an area big enough to satisfy the request (rounded up to the nearest minimum allocatable block size). If the area is bigger than the request, the kernel allocates the amount requested from the top of the area. The kernel reduces the size value in the area's information structure by the requested amount, but does not need to alter the linked list.

If there is no single free memory area in that linked list large enough to satisfy the request, the kernel returns to the table of colour node structures, and continues walking through it to find another area of the desired colour. This allocates memory of the desired colour, using high priority memory of that colour first.

An "uncoloured" memory allocation request is a request for memory of any colour, provided the memory priority code for the area is not zero. In this case the kernel walks the list of colour nodes without regard for the colour number in the descriptor, and so allocates the memory by priority only. The search stops if the colour node has a priority value of zero - such memory areas must not be allocated by an uncoloured memory request, and all further descriptors in the list will also have a priority of zero.

If the area is exactly the right size for the request, the kernel simply removes the area from the linked list.

Note: memory for the primary data space of a process, allocated by the kernel by the **F$Mem** system call when forking the program, is a special case. It is allocated from the *bottom* of a memory area.

3.4.3 Inter-task Memory Protection

Memory management hardware (usually) performs two functions. It translates logical memory addresses generated by the processor into different physical addresses to the memory chips, and it also provides protection against illegal memory accesses by generating a "bus error" signal to the processor if a particular memory access is not permitted. An access may be forbidden either because the operating system has set the "memory map" for the currently executing program in the memory management hardware to exclude that area of memory, or because the program has attempted an operation which the memory map specifies is not permitted (such as writing to an area that has been marked as "read only" in the memory map).

By changing the memory map in the memory management hardware for each program that runs, the operating system can prevent programs accessing memory that has not been specifically allocated for their use. This prevents system corruptions or crashes due to incorrect memory accesses resulting from programming errors.

OS-9 does not use the address translation capability of memory management hardware, because OS-9 is capable of running without the need for such hardware. But OS-9 can use memory management hardware to provide inter-task (that is, "between programs") memory protection. This feature of OS-9 has been available as an option for use with any memory management hardware from OS-9 version 2.2 onwards, and is standard with OS-9/68030 and OS-9/68040 (because these processors have a built-in memory management unit).

To avoid dependence on a particular memory management unit (MMU) chip, the kernel does not contain any functions to handle the MMU. Instead, these functions are provided in the System Security Module **ssm**. If **ssm** is not in ROM or the boot file (so it is not present during the kernel's coldstart), or is not in the list of "kernel customization modules" in the **init** configuration module, the kernel assumes that there is no MMU, and does not implement inter-task memory protection. Therefore to disable inter-task memory protection it is only necessary to leave **ssm** out of the boot file (and ensure it is not in ROM).

When a process is first forked, its memory map contains only its program module and its primary data space (static storage and stack). If the process allocates additional memory, that memory is added to the memory map of the process. (The program is the body of instructions in the program module. The process is this "incarnation" of the program, with its own data space and other resources). Similarly, if the process subsequently de-allocates the memory, it is removed from the process's memory map.

OS-9 can allocate memory logically in multiples of its "process minimum allocatable block size" - at present, 16 bytes. However, most memory management units cannot manage memory in such small blocks. The memory maps within an MMU will be built in larger blocks - 4k bytes is a typical size. This block size is known to OS-9 as the "system minimum allocatable block size". Physically the kernel must allocate memory to a process in multiples of this block size. However, this could waste a lot of memory if the program were to make a number of small memory requests, and the kernel allocated a separate block for each request.

Therefore, if a program makes a small memory request the kernel allocates (as it must) a whole block. But it adds the remainder of the block to the linked list of free memory fragments belonging to the process. If the program makes another small memory request, the kernel will try to use memory from the free fragments of the already allocated blocks. Only if the program makes a memory request that cannot be satisfied from the allocated blocks does the kernel allocate a new block (or blocks).

The kernel uses the same mechanism for managing memory requests from operating system components - it keeps track of the free fragments in the blocks allocated to the operating system. This is particularly important in saving memory, because the kernel itself allocates many relatively small memory areas for use as tables and resource management structures.

Note that if inter-task memory protection is not being used (SSM is not present), the kernel still uses the same technique of allocating fragments of memory from a larger block. In this case the system minimum allocatable block size is set to 256 bytes.

When a process links to a module (or creates a data module in memory), the memory area of the module is added to the process's memory map, so the program can access the module (including the module header). The module attributes (write permit in the owner, group, and public fields) determine whether the process can write over the module. Data modules are therefore a

very useful mechanism for multiple programs to share a common area of memory.

If inter-task protection is used, then user state processes cannot access the registers or memory of input/output interface chips and circuits directly. This may be seen as a restriction when building a simple application that must perform some hardware control. Therefore the **ssm** adds system calls to the operating system that permit a program (provided it is a member of group zero) to add specific areas of memory to its memory map. These system calls (**F$Permit** and **F$Protect**) are described in the chapter on the OS-9 System Calls.

The operating system (and system state processes and trap handlers) have unrestricted access to all memory. The **ssm** configures the MMU to suspend memory protections for processor accesses in supervisor state.

3.5 PROCESSES AND MULTI-TASKING

A process consists of a program that has been forked and has not exited (or been abnormally terminated), together with its data memory and a controlling memory structure used by the operating system to manage the process. This structure is known as a "process descriptor". Because OS-9 strongly suggests that programs be re-entrant, a single program may have any number of "incarnations" at any one time, each using the same program module, but having separate data memory. Each such "incarnation" is a separate process. The kernel maintains a separate process descriptor for each process, which it uses to control the process, and to retain information about the process. The process descriptor is described in the chapter on the OS-9 Internal Structure.

A process (or task) is created by a "fork" request to the operating system. Note that the word "task" is effectively synonymous with "process", and the two words are often used inter-changeably. The "fork" system call (**F$Fork**) returns a number, known as the process ID, that uniquely identifies the new process. The process ID is used for any system calls that communicate with or modify the behaviour of the process. Note that once a process has died its process ID may be assigned to a subsequently forked process, but no two processes will have the same ID at the same time. The process ID is always greater than 1 - zero is not used, and the System Process has the process ID of 1.

Under OS-9, each process has a "process priority" value associated with it. This value is assigned to the process when it is created ("forked"), and is

usually the same as the priority of the process that forked it (the "parent"). The priority of a process can be changed by the **F$SPrior** system call, and is the main mechanism for determining what share of the processor's time a process will get. The use of the process priority value, and the operation of the OS-9 process scheduler, are described in the chapter on Multi-tasking.

In a typical real time application, many processes work together to carry out the application. To do this they must exchange data, messages, and synchronization information. These functions - which are fully supported by OS-9 - are known as "inter-process communication". The OS-9 inter-process communication facilities are described in the chpater on Inter-process Communication.

At any given time, a process will be in one of the following states:

Active	Requesting processor time.
Waiting	Waiting for a child process to die.
Sleeping	Waiting for a timed period, or an external (hardware) event, or an inter-process communication signal.
Waiting for event	Waiting for an inter-process communication event.
Debugged	Waiting for its parent (a debugger program) to permit it to continue execution.
Dead	waiting to report its exit status to its parent.

Processor time is divided up between the currently active processes in time units known as "time slices". This is achieved by means of a hardware timer that produces processor interrupts at regular intervals known as "ticks". A time slice is one or more ticks. Most OS-9 systems use a 10ms tick, with two ticks per time slice. Two ticks per time slice are used rather than one, because the operating system cannot resolve time in units less than one tick. The kernel will assume that a part tick is a whole tick, so in effect a time slice of two ticks is actually between one and two ticks. A time slice of one tick could in reality be vanishingly small!

It is the execution of each process in turn for a short period of time that gives the appearance of programs executing concurrently. Therefore the length of a time slice is chosen to be short enough to give an acceptable appearance of

concurrency (for the application), yet not so short that the operating system is spending too much time in scheduling the processes. Only active processes receive processor time, so no processor time is wasted. The process scheduler of OS-9 (described in the chapter on Multi-tasking) ensures that the processor time is divided equally and evenly between all the active processes, unless the user or programmer requests otherwise - OS-9 offers several different mechanisms for the user or programmer to modify the behaviour of the process scheduler. In addition, the OS-9 scheduler has certain special features to ensure the response of high priority processes in real time applications, and to improve the throughput of the I/O system. These features are fully described in the chapter on Multi-tasking.

The **procs** utility shows all processes currently existing on the system, and displays additional information about their past performance and current state.

A process is started by a "fork" system call (**F$Fork**) from another process or an operating system component. It finishes when its program "exits" (**F$Exit** system call), or when the process encounters a fatal condition (a signal or processor exception it cannot handle). The use of signals is covered in the chapter on Inter-process Communication, while processor exceptions are described in the chapter on Exception Handling. The parent process can, in its "fork" request, pass a pointer to a memory area - known as a "parameter string" - which is copied to the new process's static storage. The parent can also specify the process priority of its new child, and request that it be allocated more static storage than the minimum specified in the program's module header.

A process can also transform itself into a new process, executing a different program. This is done by a "chain" system call (**F$Chain**). This system call is very similar to a fork, but the calling process is terminated and its process descriptor is used for the new process, which therefore has the same process ID as the original process.

Any number of processes may exist at any one time. The kernel keeps track of them by using the process descriptors. It keeps track of the process descriptors by means of the "process descriptor table", which is an array of addresses of process descriptors. The process ID of a process is an index into this table. If the entry in the table is zero, that process ID is not currently in use for any process. The process descriptor table starts off small, in order not to waste memory, but is extended by the kernel if it becomes full - the kernel allocates a new table of twice the size, and copies the old table into the new table.

3.5.1 A Dead Process

Every process initially has a parent - the process that forked it. A process may be "disinherited" (lose its parent), usually because the parent dies before the child. It would be possible for a process to be disinherited without the parent dying, by reorganizing the relevant links between the process descriptors, but there is at present no system call to do this.

A process that dies and has not been disinherited returns its exit status to its parent. This can only occur when the parent makes a "wait" system call (**F$Wait**), to "wait for child to die". Therefore a "dead" process can remain hanging around indefinitely until its parent dies or executes a "wait" request. The kernel de-allocates all of the resources of a dead process (memory, I/O paths, program module), but retains the process descriptor until the parent dies or makes a "wait" request to receive the child's exit status.

This guarantees proper operation in a multi-tasking application, where it may be essential for the parent to know the exit status of the child. However, it can cause some confusion to a user, because the user expects that when he has killed a process, it has gone.

The **shell** provides two commands to manage the death of a process. The **shell** 'w' command will wait for any one child of the **shell** to die before returning to the prompt. The **shell** 'wait' command will wait for all children of that **shell** to die.

There is no system call to disinherit a child, allowing the parent to just forget about it. One way to achieve the same effect in a multi-tasking application is to fork up a process that then forks up the desired child and dies. This leaves the child disinherited (it does not become a child of the "grandparent").

3.5.2 System State Processes

Programs normally execute in 680x0 user state. The operating system components always execute in system state (the supervisor state of the processor). In user state, certain operations such as the masking of hardware interrupts are considered illegal by the processor. Also, a process may be scheduled out (stop executing, to allow another process to execute) at any point in the program, whereas scheduling is deferred while the processor is in supervisor state (permitting system calls to be indivisible).

Programs may wish to directly handle interrupts, or to prevent themselves being scheduled out during critical code fragments. It is a relatively simple matter to add a system call (in a device driver, or a kernel customization

module) to mask and unmask interrupts, and facilities exist in the operating system to modify or pre-empt the normal multi-tasking scheduling mechanism.

Another method, however, is to run the process in system state. This can be achieved by setting the "supervisor state" flag in the program module attributes byte in the module header. When the program is forked, the operating system executes the process in system state, and it has all the privileges of the operating system, such as deferred scheduling.

However, it also has all the responsibilities of the operating system. A thorough understanding of the operation of the operating system is recommended before writing system state programs. For example, many of the system calls (especially the I/O calls) operate somewhat differently if the call is made from system state than if it is made in user state.