CHAPTER 14

OS-9 INTERNAL STRUCTURE

OS-9 is a true multi-tasking operating system. Therefore it has many resource allocation and management jobs to perform. To keep track of resources, both allocated and free, it uses a number of different data structures in memory. Also, because OS-9 is aimed at a very wide range of applications, it has very few limitations on the number of each type of resource it can manage. Therefore, to avoid wasting memory, memory is dynamically allocated as required for these data structures.

The purpose of this chapter is to identify all of the important data structures used by the operating system, and to describe in detail all of the fields of the principal structures. Because these structures determine all of the resource control mechanisms of OS-9, an understanding of these structures gives a complete understanding of how the operating system allocates and controls all of the system resources.

The OS-9 kernel is written in assembly language. Therefore all of the data structures have been defined in assembly language files. Microware has used the technique of declaring all of the symbols in the structures as public symbols, assembling the files to ROFs, and merging them to the library 'LIB/sys.l'. The **make** file to do this is provided by Microware in 'DEFS/makefile'.

This library is used by the linker when creating the kernel. For this reason, these definitions files are absolutely definitive. They are not just descriptive, but have actually been used to create the kernel. Microware supplies all of these files with OS-9, in the 'DEFS' directory.

File	Structures defined
sysglob.a	System Globals data structure.
sysio.a	Device Table, Interrupt Polling Table, and Path Descriptor data structures.
iodev.a	Device Static Storage structure for the kernel.
scfdev.a	Device Static Storage structure for SCF
rbfdev.a	Device Static Storage structure for RBF
funcs.a	System call codes and error numbers.
io.a	Path Descriptor Options Section data structures.
module.a	Module header structures.
process.a	Process Descriptor data structure.

Most of the dynamically allocated tables (arrays) used by the kernel are dynamically extendible if the table becomes full. The kernel allocates a new table twice the size of the old table, copies the old table to the first half of the new table, and frees the old table's memory. This allows small tables to be allocated initially, to conserve memory on small systems.

Some tables are not dynamically extendible. The size of each of these tables is specified in the **init** configuration module, so the implementor can tailor the table size to the system requirements. All of these non-extendible tables relate to physical resources (such as devices) which cannot be dynamically created, so the required table size is known when the computer is designed.

14.1 THE SYSTEM GLOBALS

The operating system must have some way of finding the data structures that have been created. It does this through a root structure of which only one instance exists, known as the System Globals. This structure contains pointers to other structures, which in turn may contain pointers to other structures, and so on. It is impossible to know the addresses of any of the data structures in advance, as OS-9 imposes no constraints on the system memory map.

On coldstart there must be some mechanism, suitable for all systems, of locating memory where the System Globals structure can be built. There must also be a mechanism whereby the kernel can find the address of the System Globals whenever a system call is made, or an interrupt occurs. OS-9 uses a fixed feature of the 68000 family of processors - the Reset Stack

Pointer entry of the vector table. This is the first entry in the vector table, which is situated at address zero in all 68000 systems and most other systems. The higher members of the family have a Vector Base Register to point to the vector table. This is set to zero when the processor is reset, and most systems leave it unchanged. However, OS-9 for the 68020/030/040 does fetch the pointer relative to the Vector Base Register, so it is compatible with all configurations.

The Reset Stack Pointer is assumed to point to the middle of an 8k block of RAM. This is a reasonable constraint on the implementor, as the Reset Stack Pointer *must* point to some RAM, for the boot program to work. The upper half is used to store the System Globals. The lower half is the initial stack for the boot program. At the bottom of the 8k block is the OS-9 Exception Jump Table, described in the chapter on Exception Handling (about 2.5k bytes). Note that if the boot ROM uses the "CBOOT" code supplied by Microware, the total size of this block may exceed 8k (the top part, pointed to be the Reset Stack Pointer, is still 4k bytes).

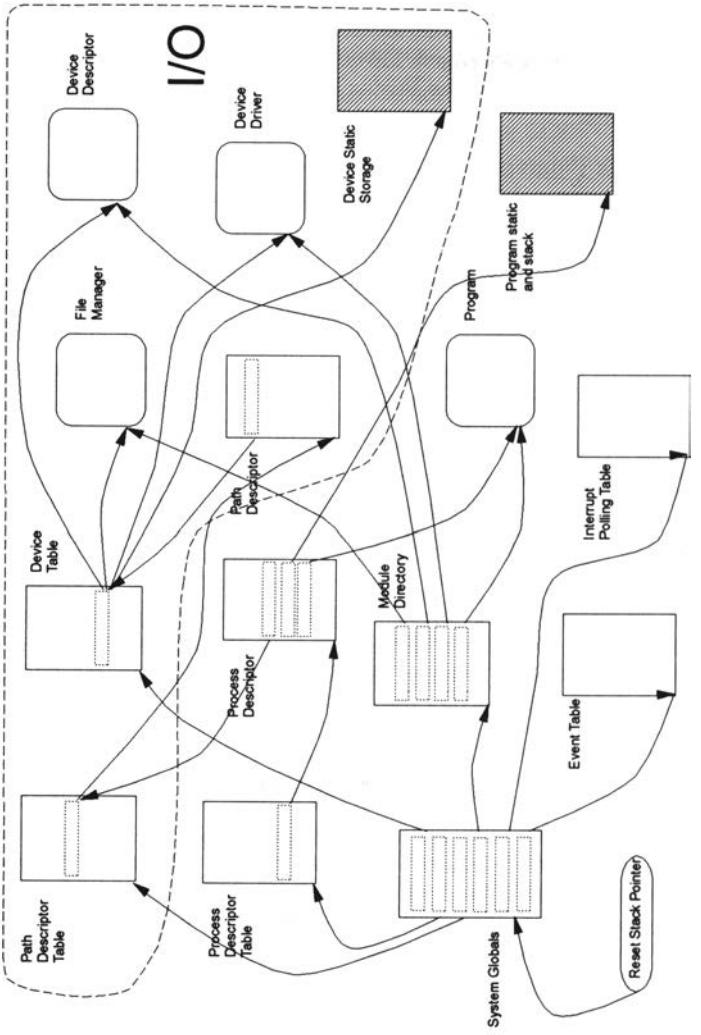
Although 4k bytes is reserved for the System Globals, the structure itself is not that big. This reservation allows for expansion of the System Globals structure without modification to the boot program. The remainder is used for the "system abort stack" (see the description of the **D_AbtStk** field in the System Globals below).

All operating system tables and memory structures are dynamically allocated from free memory, using the **F$SRqMem** system call. Root pointers to tables and linked lists of memory structures are held in the System Globals. The System Globals structure is described in detail below.

14.2 THE OTHER SYSTEM MEMORY STRUCTURES

The following sections describe all of the principal memory structures used by the kernel. In the structure description tables, the "size" field shows the length of each item as 'l' for a long word, 'w' for a word, and 'b' for a byte. If more than one item is covered by one name - for example, an array - the size letter is followed by the number of items. Figure 7 on the next page shows the interconnections between the principal types of memory structure used by OS-9. The structures contain pointers giving the base addresses of other structures.

Even though not all types of structure are shown, the diagram is rather complex! Operating system structures are shown as ordinary rectangles, modules are shown as rectangles with rounded corners, and memory areas



• **Figure 7 - OS-9 Memory Structures**

whose structures are not completely defined by the kernel are shown as shaded rectangles. The top and right part of the diagram shows the structures used by the I/O system.

14.2.1 Process Descriptor

A process descriptor is allocated for each process. A process descriptor is 2k bytes in size. Approximately the first 1k bytes contain the process descriptor memory structure, for managing the process. The remainder is used for the stack during system calls made by that process. When the process dies the process descriptor memory is freed. The kernel keeps an array of addresses of process descriptors, known as the Process Descriptor Table. The process ID number is a direct index into this table.

The structure of the process descriptor is described in detail in the Process Descriptor section of this chapter.

14.2.2 Path Descriptor

A path descriptor is allocated for each path. Note that *duplications* of a path do not create a new path descriptor, only a new process local path number to the same descriptor. A path descriptor is 256 bytes in size, cleared to zeros when first allocated. The first 128 bytes are used for storing variables for managing the path. The first part of this structure is common to all paths, and is determined by the kernel. The remainder of the 128 bytes is for use by the file manager appropriate to the particular device, and its structure varies from one file manager to another. The other 128 bytes contain the "options section", initially copied from the options section of the device descriptor used to open the path. The structure of the options section is determined by the file manager.

The kernel keeps an array of addresses of path descriptors, known as the Path Descriptor Table. The "system path number" is a direct index into this table. Note that there is a unique system path number for each open path, but individual processes refer to their paths using "local path numbers", which have a range from 0 to 31. When a process opens or duplicates a path it is assigned the first free number in its range of local path numbers as a reference to the path. This local path number is translated by the kernel through a table in the process descriptor into the system path number that uniquely identifies the path. When a path is closed, the process's local path number is freed. When all duplications of the path have been closed, the path descriptor memory is freed.

The definitions for the path descriptor shown below are taken from the file 'DEFS/sysio.a'.

Offset	Name	Size	Description
$000	PD_PD	w	System path number of this path.
$002	PD_MOD	b	Mode flags for the mode in which the file was opened (read, write, execute, directory, non-sharable).
$003	PD_CNT	b	Number of local paths (duplications) open on this system path. This byte field is now redundant, having been replaced by the word field **PD_COUNT**. However, the kernel still maintains it, for backward compatibility.
$004	PD_DEV	l	Address of the device table entry for the device this path is open on. This field forms the link between a logical path and a physical device. The device table entry contains the addresses of the file manager module, device driver module, device descriptor module, and device static storage for the device.
$008	PD_CPR	w	Process ID of the process currently making a system call on the path. The kernel sets this field at the start of a system call on this path, and clears it at the end of the system call. If this field is zero, there is no system call currently being executed on the path.
$00A	PD_RGS	l	Address of the register stack frame of the process making a system call on this path. This address is used to read the parameters of the system call (they are passed in the caller's registers), and to return results, by modifying the register images in the stack frame.
$00E	PD_BUF	l	Address of a data buffer. This field is not used by the kernel. It is for communication between the file manager and the device driver. For example, RBF puts the address of the buffer to read or write in this field before calling the device driver.
$012	PD_USER	w 2	Group number and user ID of the process that opened the path.
$016	PD_Paths	l	Address of the next path descriptor in the linked list of paths open on the same device. The device static storage contains the root pointer for this linked list.
$01A	PD_COUNT	w	Number of local paths (duplications) open on this system path. This field is set to one by the kernel when the path is first opened, and incremented whenever a duplication of the path is made (another local path number is assigned to the same system path), either explicitly (by the **I$Dup** system call) or implicitly (a child inheriting paths from the parent when forked). When a local path is closed this field is decremented. When it reaches zero

Offset	Name	Size	Description
			the path is actually closed, and the path descriptor memory is de-allocated.
$01C	PD_LProc	w	Process ID of the last process to have made a system call on this path. This field is set at the same time as **PD_CPR**, but is not cleared when the system call finishes.
$01E	PD_ErrNo	l	For the private use of the file manager - used to store the most recent error number during file manager operations (used for **errno** in file managers written in C).
$022	PD_SysGlob	l	For the private use of the file manager - used by file managers written in C to save the address of the System Globals. The address of the System Globals is passed to the file manager by the kernel in the **a6** register. File managers written in C normally put the address of the path descriptor in **a6**, as this is the static storage address register assumed by the C compiler.
$026		w 2	Reserved.
$02A		b 86	For the private use of the file manager, as static storage associated with the path. The structure is defined by the file manager writer.
$080	PD_OPT	b 128	Options section. The structure is defined by the file manager writer. The kernel initializes this area with a copy of the options section of the device descriptor the path was opened on.

14.2.3 Module Directory

The module directory is an array of structures containing the address, link count, header parity, and group identifier of each module present in memory. The structure of each entry is shown below. It is taken from the file 'DEFS/module.a'. Modules are described in the chapter on OS-9 Modules, Memory, and Processes.

Offset	Name	Size	Description
$000	MD$MPtr	l	Address of the module.
$004	MD$Group	l	Module group identifier - the address of the first module in the group.
$008	MD$Static	l	Size of the memory area allocated to contain the module group.
$00C	MD$Link	w	Link count of the module.
$00E	MD$MChk	w	Check word calculated from the module header bytes.

14.2.4 Device Table

The device table is an array of structures, one for each device currently active on the system (the device has had more "attaches" than "detaches"). A device is identified by its device descriptor module, so separate device table entries are created for different device descriptors, even if they refer to the same physical device. Each entry contains the addresses of the appropriate device descriptor, device driver, and file manager modules, plus the address of the device static storage, and a device use count. The entry is deleted (the use count, device descriptor address, and device static storage address are cleared to zeros) when the use count goes to zero.

Paths are linked to physical devices by means of a pointer (**PD_DEV**) in the path descriptor to the appropriate device table entry. The structure of a device table entry is defined in the file 'DEFS/sysio.a'.

Offset	Name	Size	Description
$000	V$DRIV	l	Address of the device driver module.
$004	V$STAT	l	Address of the device static storage.
$008	V$DESC	l	Address of the device descriptor module.
$00C	V$FMGR	l	Address of the file manager module.
$010	V$USRS	w	Current device use count.

The device table is *not* dynamically extendible. Its size is taken from the **init** module.

14.2.5 Device Static Storage

A device static storage is an area of memory used to control a single physical device interface. It may service multiple logical channels (for example, a floppy disk controller controlling four floppy disk drives). The size and usage of the device static storage is determined by the file manager and device driver controlling the device.

A separate device static storage is allocated to each device table entry unless both the device port address (in the device descriptor) and the device driver are the same as in an existing entry. In that case the kernel assumes that the new device descriptor is just another description of the same physical device (an "alias"), so the kernel uses the same device static storage as in the existing entry. The kernel clears a device static storage to zeros after allocation. The memory of a device static storage is freed when all device table entries referring to that device static storage have been deleted.

The first part of the device static storage is the same for all devices. The second part is defined by the file manager used to control the device. The third part is defined by the device driver used to control the device. Because the I/O system of OS-9 is tree structured (there is one kernel calling multiple file managers, and each file manager may call multiple device drivers), the size of the device static storage is finally determined by the linker when creating the device driver module, adding the universal definitions to the file manager definitions and the device driver definitions. The size of the device static storage required is therefore set in the device driver module header **M$Mem** field by the linker.

Note that despite the similarity in names, the device static storage is very different from a process's static storage. It is a control structure associated with the device, rather than a private store of variables for a particular application. It does not have any space used for stack - the second half of the process descriptor of the calling process is used for the stack during a system call. System calls are effectively just system state subroutines executed by the calling process.

The following table describes the first (universal) part of the device static storage. The structure is defined in the file 'DEFS/iodev.a':

Offset	Name	Size	Description
$000	V_PORT	l	"Port address" - the address of the interface device. The kernel copies this field from the **M$Port** field of the device descriptor module when it allocates the device static storage. The kernel makes no use of this field, and it does not interpret the port address except to decide whether to allocate a new device static storage, as described above. File managers also do not use this field - it is for the convenience of the device driver writer.
$004	V_LPRC	w	The process ID of the last process to use the device. The kernel does not use this field. The SCF file manager sets it to the ID of the calling process on each I/O request. It may be used by SCF device drivers. For example, a serial port device driver's receive interrupt routine will typically send the "abort" signal to the last process to use the device when the "quit key" character is received.
$006	V_BUSY	w	This field is not used by the kernel. It is typically used by the file manager to prevent concurrent calls into the device driver. The file manager checks this field before calling the device driver. If it is not zero, it is the process ID of the process currently making a call into the device driver, and the file manager "I/O queues" the current process on that process. Otherwise the file manager

Offset	Name	Size	Description
			copies the caller's process ID into this field and calls the device driver. When the device driver returns, the file manager clears this field.
$008	V_WAKE	w	This field is for the private use of the device driver. It is usually used for communication between the interrupt handler of the device driver, and the main body of the device driver. If the device driver wants to wait for an interrupt, it copies the process ID of the calling process into this field and goes to sleep. On interrupt, the interrupt handler checks this field. If it is non-zero, the interrupt handler sends the wakeup signal to that process, and clears the field to zero (as a verifiable handshake to the main body of the device driver).
$00A	V_Paths	l	Address of the first path descriptor in the linked list of path descriptors of paths open on this device (maintained by the kernel).
$00E		l 8	Reserved.
$02E	V_USER		The file manager part of the device static storage starts here.

14.2.6 Process Descriptor Table

This table is an array of the addresses of all existing process descriptors. The process ID is simply an index into this table. If an entry is zero, no process exists with that ID. Process IDs start with one (zero in a process ID field is used to mean "no process"), so the first location of the process descriptor table does not contain the address of a process descriptor. Instead, the first word contains the current size of the table (in terms of long word entries), and the second word contains the size of a process descriptor in bytes - 2048. The second entry (long word) of the table points to the process descriptor of the System Process, as this is always process 1.

14.2.7 Path Descriptor Table

This table is an array of the addresses of all existing path descriptors. A system path number is simply an index into this table. If an entry is zero, no path is open with that system path number. System path numbers start with one (zero in a system path number field is used to mean "no path"), so the first location of the path descriptor table does not contain the address of a path descriptor. Instead, the first word contains the current size of the table (in terms of long word entries), and the second word contains the size of a path descriptor in bytes - 256.

14.2.8 Interrupt Polling Table

This is an array of structures, one for each hardware interrupt handler routine currently installed. The System Globals contains 199 pointers, each of which is (if not zero) a root pointer to a linked list of these structures. Therefore there is one such linked list for each interrupt vector on which one or more interrupt handler routines has been installed. Interrupt handlers are installed using the **F$IRQ** system call.

The interrupt polling table is *not* dynamically expandable - its size is set by an entry in the **init** module. Each entry is 18 bytes long, and has the following structure (defined in the file 'DEFS/sysio.a'):

Offset	Name	Size	Description
$000	Q$LINK	l	Pointer to the next entry in the linked list for this vector number.
$004	Q$SERV	l	Address of interrupt handler routine.
$008	Q$STAT	l	Address of interrupt handler's static storage - normally the device static storage. This field is not checked or used by the kernel, except to identify the entry when it is to be deleted, and to pass the static storage address to the interrupt handler.
$00C	Q$POLL	l	Port address. This field is not checked or used by the kernel, except to pass the port address to the interrupt handler.
$010	Q$PRTY	b	Polling priority. An entry with a low polling priority number precedes an entry with a higher priority number, and so is called first to service an interrupt on that vector. A priority of zero means that the entry must be the only one on this vector.
$011		b	Reserved.

14.2.9 Event Table

This is an array of structures, one for each event currently in existence. Events are created (and maintained) by the **F$Event** system call. Each entry contains the event name, event number, the event's current value, and the link count of the event.

The event table is dynamically extendible. Its initial size is set by an entry in the **init** module - it is usually zero. Each entry is 32 bytes long, and has the following structure:

Offset	Size	Description
$000	w	Event number.
$002	b 12	Event name (null terminated if not 12 bytes long).
$00E	l	Event value.
$012	w	Wakeup increment.
$014	w	Signal increment.
$016	w	Event link count.
$018	l	Address of process descriptor of first process waiting on the event - start of linked list of process descriptors.
$01C	l	Address of process descriptor of last process waiting on the event.

The position within the table is called the event index. The first structure is index zero, the second is index one, and so on. The event ID that is returned by a call to create or link to an event is a long word - the high word is the event number, the low word is the event index. The kernel keeps a record of the last event number assigned, in the high word of the **D_EvID** field in the System Globals, initially zero. The kernel increments this word in the System Globals before using it to create the event ID for a new event.

The combination of event number and index as the event ID gives a high degree of confidence that a program cannot accidently reference an event that has been deleted (the event ID will not match any existing event), while maintaining the speed of the event functions, which internally use the event index.

14.2.10 Service Dispatch Tables

The operating system calls are customizable - existing handler functions can be replaced by new handlers, and new system calls can be defined. This feature is provided through the dispatch tables, which contain the addresses of the system call handler functions. There are two tables - the system dispatch table, and the user dispatch table, for calls made from system state and user state respectively.

Each table consists of 512 long words. The first 256 long words are the addresses of the handler functions for each of 256 possible system call codes. The remaining 256 long words are the addresses of the (optional) private static storage for each of the handler functions. System call handler functions are installed by the **F$SSvc** system call, as described in the chapter on Exception Handling.

14.3 SYSTEM GLOBALS STRUCTURE

The System Globals structure is the root of all information in the system. Starting from this structure (whose address is held at memory location zero) all other memory structures can be located. The System Globals also contains system-wide variables, and system constants defined at coldstart. The System Globals structure is defined in the file 'DEFS/sysglob.a'. This section describes the function of each field (under OS-9 version 2.4).

Fields in the System Globals can be read directly by operating system components, or indirectly by user state programs using the **F$SetSys** system call (or **_getsys()** C library function). User state programs can change certain locations, again by using the **F$SetSys** system call, (or the **_setsys()** C library function). This technique must be used, rather than writing directly to the System Globals structure, as the kernel may need to take additional action when certain fields are altered. This is especially true for the fields that modify the behaviour of the process scheduler, as described in the chapter on Multi-tasking.

The table below shows the System Globals structure. The symbol names are defined in the file 'sysglob.a'. The named items are not necessarily contiguous. Microware has reserved several fields for future use, while some fields are skipped to keep word and long word fields on even addresses, and some fields are historical relics (from previous versions of OS-9) that are no longer used.

Offset	Name	Size	Description
$000	D_ID	w	Set by the kernel to the module sync code $4AFC after coldstart has finished. If set to $6F6B (ASCII "ok"), the kernel does not check module CRCs during coldstart, effecting a "warm start".
$002	D_NoSleep	w	Set non-zero to prevent the system process from sleeping.
$020	D_Init	l	Set by the kernel to the address of the configuration module **init**, to speed access.
$024	D_Clock	l	Set by the kernel to the address of its clock tick handler routine. The clock driver calls this routine every tick interrupt.
$028	D_TckSec	w	Set by the clock driver to the number of ticks per second.
$02A	D_Year	w	The current year (for example, 1993). This field and the following month and day fields are not dynamically maintained by the kernel. They are updated only when a

Offset	Name	Size	Description
			program requests the current date and time, using the **F$Time** system call.
$02C	D_Month	b	The current month (1 to 12).
$02D	D_Day	b	The current day in the month (1 to 31).
$02E	D_Compat	b	This is the first of two fields of bit flags used to modify the kernel behaviour to maintain the behaviour of previous versions of OS-9, or to cope with unusual hardware configurations (see below).
$02F	D_68881	b	Floating Point Unit type (68020/030/040 systems only) 0 no FPU 1 68881 2 68882 40 68040
$030	D_Julian	l	Julian day number - maintained by the kernel's tick handler routine. See the chapter on OS-9 System Calls for a full description of Julian dates.
$034	D_Second	l	System time, as seconds left until midnight - maintained by the kernel's tick handler routine. Held in this unusual format to speed up the tick handler, which needs only to decrement this field and test for zero (new day).
$03A	D_IRQFlag	b	Kernel flag - currently servicing an interrupt. This field is initialized to $FF on coldstart. On entry to the kernel's interrupt handler the field is incremented - if it is now zero, the stack pointer register is set with the value in the **D_SysStk** field, so switching to the "interrupt stack" if an interrupt occurs unless an interrupt is already being serviced. On exit from the kernel's interrupt handler this field is decremented.
$03B	D_UnkIRQ	b	The number of times an unknown interrupt has occurred in a row. If no interrupt handler acknowledges ownership of an incoming interrupt, the kernel increments this field. If an interrupt is handled successfully the kernel clears this field. If the field increments to zero (count of 256), the kernel masks interrupts to the level of the interrupt that cannot be handled. This mechanism is attempts to cope with hardware problems or hardware configuration errors.
$03C	D_ModDir	l 2	Address of the module directory, and address of end of module directory memory plus one (to speed up module directory searches).
$044	D_PrcDBT	l	Address of the Process Descriptor Table.
$044	D_PthDBT	l	Address of the Path Descriptor Table.
$04C	D_Proc	l	Address of the Process Descriptor of the Current

Offset	Name	Size	Description
			Process. The Current Process is the process currently executing. A system call routine will use this field to gain access to the process descriptor of the calling process. The Current Process is *not* in the Active Queue.
$050	D_SysPrc	1	Address of the System Process Descriptor. The System Process (always process 1) is woken by the kernel's tick handler. Its function is to wake up processes in timed sleep, and to send alarms to processes.
$054	D_Ticks	1	Ever incrementing tick count. Indicates the time since coldstart. Useful for timing intervals.
$058	D_FProc	1	Address of the process descriptor of the process whose context is in the FPU registers. The kernel avoids unnecessary saving and reloading of the FPU registers (the FPU context can be very large) if other processes have not used the FPU (their FPU context shows "idle").
$05C	D_AbtStk	1	System state abort stack pointer. This is intended to allow a graceful abort from a bus error within an interrupt handler. The abort stack is the remaining memory after the System Globals structure up to the 4k bytes reserved for the System Globals. On coldstart this field is initialized to point to the last long word of the abort stack ($0FFC from the start of the System Globals). The address of the kernel's bus error handler function is written to that last long word of the abort stack. On entry to the kernel's interrupt handler the kernel decrements this field by 4 (allocating a long word on the abort stack), and writes its current (interrupt) stack pointer minus 4 to this long word pointed to by the abort stack pointer. The kernel then makes a subroutine call to the device driver's interrupt handler. Therefore during the interrupt handler of a device driver this field points to a recovery stack pointer value which can be loaded into the system stack pointer, after which an **rts** instruction will to return to the kernel. On exit from the kernel's interrupt handler the kernel increments this field by 4, so ditching the saved interrupt stack pointer. Apart from maintaining this field as described, the kernel does not use it itself, even within its bus error handler.
$060	D_SysStk	1	Address of the stack memory to use during interrupts. During the kernel coldstart this field is initialized to the address of the System Globals, so the interrupt stack is the memory below the System Globals. Once the kernel has linked to the **init** module it uses the "size of interrupt stack" field to allocate a separate area of memory, whose top address plus one is placed in this

Offset	Name	Size	Description
			field (push down stack). The default value for the size of this stack when creating the **init** module is 1k bytes.
$064	D_SysROM	1	Boot ROM execution entry point - the address of a branch table in the boot ROM. This gives access to the non-interrupt driven input/output functions for the system console, which may be useful for announcing errors from within interrupt handlers.
$068	D_ExcJmp	1	Address of the Exception Jump Table.
$06C	D_TotRAM	1	Total RAM found by the boot program.
$070	D_MinBlk	1	Process minimum allocatable block size - the minimum size of memory that can be allocated and freed by the software memory management functions. Memory is always allocated in multiples of this value (currently 16).
$07C	D_BlkSiz	1	System minimum allocatable block size - the minimum size of memory that can be managed by the Memory Management Unit, if present, otherwise a default value (currently 256).
$080	D_DevTbl	1	Address of the Device Table.
$088	D_AutIRQ2	1 7	68070 on-chip I/O autovector interrupt polling table root pointers. The 68070 processor has on-chip interrupt sources not present in other members of the 68000 family.
$0A4	D_VctIRQ	1 192	Vectored interrupt polling table root pointers. The 68000 family supports vector numbers 64 to 255. The kernel subtracts 64 from the vector number to form an index into this table.
$3A4	D_SysDis	1	Address of the system state service dispatch table.
$3A8	D_UsrDis	1	Address of the user state service dispatch table.
$3AC	D_ActivQ	1 2	Active process queue pointers. The address of the first and last process descriptors in the linked list of process descriptors of active processes. These are processes that are requesting processor time (but excluding the current process - the process currently executing).
$3B4	D_SleepQ	1 2	Sleeping process queue pointers - the linked list of processes in timed and untimed sleep. This list is ordered by time before wakeup, with the untimed sleeping processes at the end of the list.
$3BC	D_WaitQ	1 2	Waiting process queue pointers - the linked list of processes waiting for a child process to die.
$3C4	D_ActAge	1	Active queue age. This is the decrementing "system age" used in the management of process scheduling.
$3C8	D_MPUTyp	1	Microprocessor in use, as an integer

Offset	Name	Size	Description
			(68000, 68010, 68020, 68030, 68040, 68070, 68300).
$3CC	D_EvTbl	l 2	Address of event table and address of end of event table memory plus one (to speed up event table searches).
$3D4	D_EvID	l	Last event number used. Only the high word of this field is used. Initially set to zero, this word is incremented before an event is created. It is used as the high word of the event ID of the event created (the low word of the event ID is the index into the event table for the event structure used for the new event).
$3D8	D_SPUMem	l	Address of the static storage for the System Security Module. If this field is zero, inter-task memory protection is not in use (SSM is not active).
$3DC	D_AddrLim	l	Highest memory address found during startup (both RAM and ROM).
$3E0	D_Compat2	b	This is the second of two fields of bit flags used to modify the kernel behaviour to maintain the behaviour of previous versions of OS-9, or to cope with unusual hardware configurations (see below).
$3E1	D_SnoopD	b	The kernel sets this field non-zero if all processor memory data caches are coherent ("snoopy") or no data caches exist. That is, the caches do not need flushing after another bus master (for example, a DMA controller) has written to memory. The internal caches of a 68040 are normally snoopy. Other caches are usually not snoopy.
$3E2	D_ProcSz	w	The size of a process descriptor (currently 2k bytes). This value should be used to calculate the size of the system state stack, if system state stack checking code is being written.
$3E4	D_PolTbl	l 8	Autovectored interrupt polling table root pointers. The 68000 family supports 7 autovectors (one for each interrupt level). The autovector number is 24 plus the interrupt level of the incoming interrupt, 1 to 7. The kernel subtracts 24 from the vector number to generate an index into this table. The first location is therefore never used (OS-9 does not provide support for the "Spurious Interrupt" exception).
$404	D_FreeMem	l 2	Address of the first and last entries in the linked list of memory area descriptors in the memory colour node table. As there is no system call to dynamically introduce new memory areas to the system, the colour node table (built at coldstart) is never changed.

Offset	Name	Size	Description
	The next three fields are not currently used. They were intended for use in a multi-processor version of OS-9:		
$40C	*D_IPID*	*w*	*Inter-processor identification number.*
$410	*D_CPUs*	*l*	*Address of the array of processor descriptor list heads.*
$414	*D_IPCmd*	*l 2*	*Start and end addresses of the inter-processor command queue.*
	The next three fields are used only by the **F$CCtl** system call installed by the **syscache** kernel customization module (described in the chapter on OS-9 System Calls):		
$764	D_CachMode	l	68020/68030/68040 Cache Control Register current setting in abstracted form.
$768	D_DisInst	l	Instruction cache disable request depth - permits nested calls to disable the instruction caches. If non-zero, the instruction caches are currently disabled.
$76C	D_DisData	l	Data cache disable request depth - permits nested calls to disable the data caches. If non-zero, the data caches are currently disabled.
$770	D_ClkMem	l	Address of a clock tick thief's static storage. By substituting the address of its own tick handler in the field **D_Clock**, an operating system component can be called on every tick interrupt. Its handler must finish with a jump to the kernel's tick handler, whose address was saved from the original value in **D_Clock**. This requires that the handler have some static storage. The address of that static storage can be written here. System state alarms make this technique redundant for almost all purposes.
$774	D_Tick	w	Number of ticks remaining in the current second. This field is initialized by the kernel from the field **D_TckSec**, and is decremented by the kernel's tick handler. When it reaches zero the tick handler re-initializes the field, and decrements the field **D_Second**.
$776	D_TSlice	w	Ticks per time slice (copied by the kernel from the **init** configuration module).
$778	D_Slice	w	Number of ticks remaining in the current time slice. The kernel initializes this field from **D_TSlice** when it starts a time slice for a process. The kernel's tick handler decrements this field. When the field reaches zero, the tick handler sets the "timed out" flag in the process descriptor of the current process (see **D_Proc**).
$77C	D_Elapse	l	Number of ticks remaining before the system process should be woken. When a process requests a timed sleep

Offset	Name	Size	Description
			or an alarm signal, the kernel determines whether it will be the first process to need a wakeup or a signal. If so, it sets this field to the number of ticks to elapse before the system process needs to take action. The kernel's tick handler decrements this field (if it is not zero), and wakes up the system process when it reaches zero. The system process performs the necessary wakeup(s) and sends the necessary signal(s), and then recalculates the time to the next wakeup or alarm, setting this field accordingly (or to zero if none).
$780	D_Thread	l 2	Addresses of the first and last thread queue linked list entries in the thread list, for the list of immediate or absolute time alarm signals. The kernel makes an entry in this list (allocating a thread structure from system memory) when a process requests an alarm signal by an absolute time, or which the kernel calculates requires an immediate signal. The entries are linked, ordered by the time of the alarm.
$788	D_AlarTh	l 2	Addresses of the first and last thread queue linked list entries in the thread list, for the list of relative time and cyclic alarm signals. The kernel makes an entry in this list (allocating a thread structure from system memory) when a process requests an alarm signal by a relative time, or at repeating intervals. The entries are linked, ordered by the time of the alarm.
$790	D_SStkLm	l	Interrupt stack memory base address.
$794	D_Forks	l	Number of processes in existence.
$798	D_BootRAM	l	Total amount of RAM found by the boot program.
$79C	D_FPUSize	l	Maximum size of FPU saved state frame.
$7A0	D_FPUMem	l	Address of static storage for the FPU emulator (for 68040).
$7A4	D_IOGlob	b 256	This area of memory is for use by implementation-dependent operating system components, such as device drivers. Microware have reserved the first 32 bytes. This memory should only be used for variables that can only occur once in the system, such as images of processor board write-only registers. The structure of this memory is defined in 'DEFS/ioglob.a', which the implementor should edit as necessary (and then remake the 'LIB/sys.l' library).

The following three locations are used to modify the behaviour of the kernel's process scheduler (see the chapter on Multi-tasking). They can be altered by means of the **F$SetSys** system call (and must not be modified directly):

$8A6	D_MinPty	w	Minimum process priority - processes with a lower

Offset	Name	Size	Description
			priority receive no processor time.
$8A8	D_MaxAge	w	Maximum process priority - processes with a higher priority execute strictly in priority order. Processes with a lower priority are scheduled normally, but receive no time if a high priority process is active.
$8AA	D_Sieze	w	*(sic)* If this field is not zero it is assumed to be a process ID. The kernel will not give time to any other process until this field is changed.
$8AC	D_Cigar	l	The bytes of this field contain version numbers for the more important of the memory structures:
$8AC		b	System Globals version - currently 1
$8AD		b	Process descriptor version - currently 1
$8AE		b	Module directory entry version - currently 1
$8AF		b	Module header version - currently 1
$8EC	D_SysDbg	l	Address of the entry point of the ROM based debugger. A user program (such as the 'break' utility) can call the ROM based debugger using the **F$SysDbg** system call. This field is initialized during the kernel coldstart to the "boot entry point" address passed by the boot program, plus 16.
$8F0	D_DbgMem	l	The System State Debugger program uses this field to save the address of its static storage. A non-zero value indicates that the debugger is active.
$8F8	D_Cache	l	Address of the RBF disk cache buffer header. A non-zero value indicates that disk caching is active.

The **D_Compat** and **D_Compat2** fields contain bit flags to modify the behaviour of the kernel, to retain compatibility with earlier versions of OS-9, and to cope with hardware peculiarities. The bit flags of **D_Compat** are:

Bit	Meaning when set
0	Save all processor registers on interrupt (instead of just a subset).
1	Don't use the 68000 **stop** instruction to wait for interrupt when no processes are active - loop in software.
2	Don't implement the "sticky modules" feature.
3	Don't enable 68030 cache burst mode (used by the **syscache** kernel customization module).
4	Fill memory with a pattern when allocated or freed.

5	Don't attempt to start the clock driver during kernel coldstart (otherwise the kernel executes the **F$STime** system call with a date of zero, to start the clock ticks and read a battery-backed time-of-day clock if one exists).

The bit flags of **D_Compat2** are:

Bit	Meaning when set
0	External instruction cache is snoopy or non-existent.
1	External data cache is snoopy or non-existent.
2	Processor instruction cache is snoopy or non-existent.
3	Processor data cache is snoopy or non-existent.
7	Kernel should not disable data caching during I/O system calls.

A snoopy (or coherent) memory cache is one that watches bus activity while another bus master is active. It automatically updates (or invalidates) its contents if the bus master writes to a memory location that is also held in the cache, and automatically inhibits the memory and supplies the cached value if the bus master tries to read a memory location that is "stale" because a "dirty" location in the cache has not yet been flushed to memory.

If the data caches are not snoopy (or non-existent), then device drivers that control devices with DMA must flush the caches before writing to the device or after reading from the device.

14.4 PROCESS DESCRIPTOR STRUCTURE

The Process Descriptor contains all the variables needed to control a process and record its resource allocations. It also contains the memory to be used as stack space during system calls made by the process. This removes the need for the program to reserve space in its stack for use by the operating system. The Process Descriptor structure is defined in the file 'DEFS/process.a'. This section describes the function of each field (under OS-9 version 2.4).

Offset	Name	Size	Description
$000	P$ID	w	Process ID of this process.
$002	P$PID	w	Process ID of parent process. This field is zero if the parent has died.

Offset	Name	Size	Description
$004	P$SID	w	Process ID of next sibling process - forms a chain of processes that are children of the same parent. This field is zero if this process is the last in the chain for this parent (or it is the only child of the parent).
$006	P$CID	w	Process ID of the first child process. The **P$SID** field of the first child process holds the process ID of the second child process, and so on. In this way the kernel keeps track of all the children of a process. This field is zero if the process has no children.
$008	P$sp	l	System stack pointer, saved on the last system call made from user state, after the processor registers have been stacked. During a system call this field points to a stack frame of the processor registers of the calling user state program, while the **a5** register points to the stack frame of the caller, whether system or user state.
$00C	P$usp	l	User stack pointer, saved on the last system call made from user state. During a system call this field points to the stack of the calling user state program.
$010	*P$MemSiz*	*l*	*Not used.*
$014	P$User	w 2	User group number and user ID of the user who forked the process (may be changed using the **F$SUser** system call).
$018	P$Prior	w	Process priority.
$01A	P$Age	w	Process "age". This field is not used by the kernel, but is calculated when a copy of the process descriptor is requested using the **F$GPrDsc** system call. If the process is not active it is set equal to the process's priority. Otherwise it is set equal to the process's scheduling constant minus the current system age (**D_ActAge**), which is an indication of the number of processes that have been put into the active queue (including rescheduling the current process when its time slice expires) since the process itself was put in the active queue. If the calculated value exceeds 10000 it is assumed to be "unreasonable", and a value of -1 ($FFFF) is set. If the "maximum age" field in the System Globals **D_MaxAge** is not zero, and the calculated age of this process is greater than or equal to the "maximum age", the age is set to the maximum age minus one. None of this affects the scheduling of the process.
$01C	P$State	w	Only the high byte of this field is used. This byte is a set of bit flags indicating the current state of the process. The flags are described below.
$01E	P$Task	w	Process "task number" - for use by an SSM controlling an MMU that can store multiple memory maps at the

Offset	Name	Size	Description
			same time.
$020	P$QueuID	b	This field contains an ASCII printable character, indicating which queue (linked list) the process descriptor is currently in. The possible values are listed below.
$021	P$SCall	b	The function code of the last system call executed in user state.
$022	P$Baked	b	A kernel check flag - non-zero if the process was created by the **F$Fork** system call.
$024	P$DeadLk	w	ID of the process to which an I/O deadlock has been lost.
$026	P$Signal	w	Signal code of pending signal. This is the code of the most recently received signal. (Note: prior to OS-9 version 2.4, this was the *oldest* signal, that is, the. signal to be handled first.) This field is zero if there is no signal pending.
$028	P$SigVec	l	Address of process's signal handler function. If this field is zero, the process has not installed a signal handler function - the kernel will kill the process if a signal is received.
$02C	P$SigDat	l	Address of the data space of the signal handler function - normally the process's static storage.
$030	P$QueueN	l	Address of the process descriptor of the next process in the queue (linked list). This field and the following field provide the links for a doubly linked list of process descriptors. Depending on the value in **P$QueuID**, this will be the active queue, the sleeping queue, the waiting queue, or an event queue.
$034	P$QueueP	l	Address of the process descriptor of the previous process in the queue.
$038	P$PModul	l	Address of the program module that was forked for this process.
$03C	P$Except	l 10	Addresses of the process's handler functions for the "hardware exceptions" (bus error, address error, illegal instruction, and so on). If a field is zero, the process has not installed a handler for the corresponding exception. In that case, the kernel will kill the process if that exception occurs while the process is executing in user state. See the chapter on Exception Handling.
$064	P$ExStk	l 10	Addresses of the static storage areas in which to build a stack frame of the processor registers if a "hardware exception" occurs. If a field is zero, the kernel will build the stack frame on the process's user state stack when the corresponding exception occurs.

Offset	Name	Size	Description
$08C	P$Traps	l 15	Addresses of the trap handler modules installed to handle **trap #n** instructions 1 to 15. If a field is zero, the process has not installed a trap handler module (using the system call **F$TLink**) for the corresponding **trap #n** instruction.
$0C8	P$TrpMem	l 15	Addresses of the static storage memory areas allocated for the trap handler modules. If a field is zero, the trap handler's module header shows the trap handler needs no static storage, or no handler is installed for that **trap #n** instruction.
$104	P$TrpSiz	l 15	Sizes of the static storage areas allocated for the trap handler modules.
$140	P$ExcpSP	l	System state recovery stack pointer - value to place in the system stack pointer if a "hardware exception" occurs while this process is executing in system state. Note: if a hardware exception occurs while an interrupt is being serviced, the kernel regards this as a fatal error, and reboots the system.
$144	P$ExcpPC	l	Address of the system state hardware exception handler. The kernel causes execution to divert to this address if a "hardware exception" occurs while the process is executing in system state. If a hardware exception occurs while the process is executing in system state and this location is zero, the kernel regards this as a fatal error, and reboots the system. At the start of a system call the kernel initializes this field and **P$ExcpSP** to values that simply cause an early termination of the system call, with an abort of the process. At the end of a system call the kernel restores the previous values (usually zero). An operating system component (such as a device driver) can substitute its own values during a call, in order to take more appropriate action on exception.
$148	P$DIO	b 32	An area in which to store information about the current data and execution directories. The first 16 bytes are for information about the current data directory. The other 16 bytes are for information about the current execution directory. In each area the kernel uses the first 4 bytes to store the address of the device table entry for the device on which the directory resides, and reserves the following two bytes. The remaining 10 bytes are for use by the file manager controlling the device, as shown in the file 'DEFS/sysio.a'.
$168	P$Path	w 32	Table of system path numbers for open paths. When the process opens (or inherits) a path it is given a local path number in the range 0 to 31. The local path number is

Offset	Name	Size	Description
			an index into this table, giving the system path number for the path, which identifies the appropriate path descriptor. If a field is zero, the process does not have a path open with that local path number.
$1A8	P$MemImg	1 32	Table of addresses of memory areas allocated by the process, including the static storage ("primary data area"), and memory allocated by user state trap handlers (excluding the static storage of the trap handlers). When the process makes a memory allocation request, the kernel saves the address of the allocated memory in this table. The process can have up to 32 outstanding memory allocations at any one time. If a table entry is zero, there is no memory allocation corresponding to that table entry. The table is always ordered by memory address (low addresses first). If an entry is erased (because the memory is returned), the entries above are moved down, so the table has no "holes". If two or more allocated areas are contiguous, they are coalesced into one table entry.
$228	P$BlkSiz	1 32	Table of sizes of allocated memory areas. The size is the actual size allocated from the system free memory pool as the result of a memory request, and so is always a multiple of the system minimum allocatable block size. The area may contain free fragments that have not yet been given to the program, which is able to allocate memory in multiples of the process minimum allocatable block size (which is smaller than the system minimum allocatable block size). The kernel keeps track of these fragments - which belong to the process, but have not yet been allocated to a program request - through the linked list rooted in **P$frag**.
	The next three fields are used only if the process was created by the **F$DFork** system call (a debugged process). The process is then under the control of its parent (the debugger):		
$2A8	P$DbgReg	1	Address of the register stack frame buffer in the parent's static storage. The kernel copies the process's registers to this stack frame after an **F$DFork** or **F$DExec** system call, and sets the process's registers from this stack frame when an **F$DExec** system call is made by the parent.
$2AC	P$DbgPar	1	Address of the parent's process descriptor. If this field is zero, the process is not a "debugged" process.
$2B0	P$DbgIns	1	Total number of instructions executed in user state so far within an **F$DExec** system call from the parent. This field is not used if the **F$DExec** call specified "full speed" execution.

Offset	Name	Size	Description
$2B4	P$UTicks	1	Number of tick interrupts that have occurred while this process was the current process in user state.
$2B8	P$STicks	1	Number of tick interrupts that have occurred while this process was the current process in system state.
$2BC	P$DatBeg	1	Date (in Julian form) when the process was forked.
$2C0	P$TimBeg	1	Time (seconds since midnight) when the process was forked.
$2C4	P$FCalls	1	Number of system calls (other than I/O system calls) made in user state.
$2C8	P$ICalls	1	Number of I/O system calls made in user state.
$2CC	P$RBytes	1	Number of bytes read (**I$Read** and **I$ReadLn**) in user state without error.
$2D0	P$WBytes	1	Number of bytes written (**I$Write** and **I$WritLn**) in user state without error.

The next two fields are used in building a queue of processes waiting for an I/O resource (such as a path or device). The kernel or file manager, on finding that a requested I/O resource is already in use by another process, will add this process to the I/O queue on that process using the **F$IOQu** system call. When the kernel is finishing an I/O system call, it checks the process descriptor to see if it has an I/O queue (**P$IOQN** is not zero). If so, it wakes up the first process in the queue (the process ID in **P$IOQN**).

The queue is ordered by the scheduling constants of the processes at the time they were put in the queue. A process being put in the queue is inserted after other processes with the same or greater scheduling constants (see the chapter on Multi-tasking).

$2D4	P$IOQP	w	The process ID of the previous process in the I/O queue this process is waiting in. If this field is zero, this process is not I/O queued.
$2D6	P$IOQN	w	The process ID of the next process in the I/O queue. If the **P$IOQP** field is zero, this process has the I/O resource, and this field is the process ID of the process that will be woken when the resource becomes free.
$2D8	*P$Frags*	*1 2*	*Not used (historical, from before coloured memory).*
$2E0	P$Sched	1	Scheduling "constant". This field is calculated when the process is put into the active queue. It is the sum of the system age (**D_ActAge**) at that time and the process's priority. It determines the position of the process in the active queue. See the chapter on Multi-tasking for a full description.
$2E4	P$SPUMem	1	Address of memory allocated by the System Security Module to contain the process's memory map as required for the MMU. This field is zero if the SSM is not used.

Offset	Name	Size	Description
	The next two fields are set by the kernel as part of the **F$DExec** system call made by the debugger to request that the process execute one or more instructions. These fields are not used if the process is not a "debugged" process (created by **F$DFork**).		
$2E8	P$BkPtCnt	1	Number of breakpoints set. The **F$DExec** system call specifies a number of breakpoints for the kernel to set in the program being debugged.
$2EC	P$BkPts	w 16	If "full speed" execution was requested by an **F$DExec** system call, the breakpoints are set by the kernel by writing an illegal instruction at each breakpoint location. The instruction word that was at each breakpoint location is saved in this table by the kernel, and restored when the program halts (breakpoint, exit, or hardware exception). Otherwise (trace mode execution) the kernel sets the process into trace mode, so the process halts after each instruction with a trace exception. The kernel then compares the process's program counter with each breakpoint address in the list specified in the **F$DExec** call.
$30C	P$Acct	1 8	This space is available for use by a "user accounting module". A user accounting module is a kernel customization module that is called whenever a process is forked or terminated, to keep track of the use of resources by users.
$32C	P$Data	1	Address of the process's static storage ("primary data area") allocated by the kernel when the process is forked. Also the first entry in the **P$MemImg** table. It is this memory area that is expanded (or contracted) by the **F$Mem** system call.
$330	P$DataSz	1	Size of the process's static storage, including the stack, but excluding the parameter string.
$334	P$FPUSave	1	Address of the memory area (allocated when the process is forked) in which to save the register frame and context of the FPU when the process ceases to be the current process. If the system does not have an FPU, this field is zero.
	The next two fields have a similar purpose to the fields **P$Except** and **P$ExStk**. They are used for the additional "hardware exceptions" that can be generated by the FPU. If the system does not have an FPU, these fields are not used.		
$338	P$FPExcpt	1 7	Addresses of the process's handler functions for the "FPU exceptions" (divide by zero, not a number, and so on).
$354	P$FPExStk	1 7	Addresses of the static storage areas in which to build a stack frame of the processor registers if an "FPU exception" occurs. If a field is zero, the kernel will build the stack frame on the process's user state stack when

Offset	Name	Size	Description
			the corresponding exception occurs.
$370	P$SigLvl	b	Signal mask nesting level. If this field is non-zero, signals are masked for the process. In this case, if a signal is received the process is still made active (if it was not already), and the signal is put in the signal queue, but the process's signal intercept handler is not called. When the process clears this field (using the **F$SigMask** system call), the signal intercept handler is immediately called for each pending signal. This field can be incremented, decremented, or cleared by the **F$SigMask** system call.
$371	P$SigFlg	b	This field is a set of bit flags for the signal mechanism. Currently, only bit 7 is used. It is set when the process receives a signal while active, and cleared when the process goes to sleep, or returns to user state. If this flag is set, a process can go to sleep even if a signal is pending. This allows a system call to go to sleep, waiting for a signal from (for example) an interrupt service routine, even though another signal is pending.
$372	P$Sigxs	w	Number of free entries in the signal queue.
$374	P$SigMask	l	This field is a set of 32 bit flags, each bit (0 to 31) corresponding to the signal code of the same number. If the bit is set, then a signal of that code sent to the process is ignored - the process is not woken, and the signal is not queued. Signals 0 (kill) and 1 (wakeup) cannot be filtered in this way. Due to the coding of the **F$Send** system call, signal code 32 will be ignored if bit zero of this field is set. There is no system call to alter this field.
$378	P$SigCnt	l	Number of signals pending.
$37C	P$SigQue	l	Address of the signal queue element containing the next signal to process (the oldest pending signal). The signal queue is a doubly linked list of structures containing one signal code each. The queue is arranged as a ring, so that the "next" pointer of the last entry points to the first entry (oldest pending signal). As each signal is processed (the process's signal handler is called), the kernel clears the signal code field of the entry, and advances this field to point to the next entry in the queue. While queue structures may be cleared (signal code field is set to zero, indicating no signal pending), they are not removed from the queue, and their memory is not freed until the process dies. This field is zero until the process receives a signal.
$380	P$DefSig	l 4	Initial signal queue structure. When the process first receives a signal this field is installed as the first and

Offset	Name	Size	Description
$380	P$DefSig	1 4	Initial signal queue structure. When the process first receives a signal this field is installed as the first and only entry in the signal queue. If the queue is full when a signal is received, a signal queue structure is allocated from system memory. Therefore if the process never receives a signal with one already pending, no dynamically allocated structures are needed.
$390	P$Thread	1 2	Addresses of the first and last "thread" structures allocated to this process. This is a doubly linked list of nominally general-purpose structures. At present it is used to record outstanding "alarms" installed by the process using the **F$Alarm** system call. If the process has no alarms outstanding these fields will be zero.
$398	P$frag	1 2	Addresses of the first and last memory fragment structures. This is a doubly linked list of structures identical to the memory colour node structures used to manage the free pool of memory. This linked list identifies the free memory fragments not yet used from the memory allocated to the process. This is necessary because the kernel will take memory from the free pool only in multiples of the system minimum allocatable block size (at least 256 bytes), but is able to allocate to the process in multiples of the process minimum allocatable block size (currently 16 bytes). When a process makes a request for memory, the kernel first attempts to satisfy the request from the fragments identified by this linked list. Only if this fails does the kernel allocate additional system memory to the process.
$3A0	P$MOwn	1	Original owner of the primary module of this process. This protects against a program, written by a user who is not a super user, modifying its own module header (which is naturally within its memory map) to set the "module owner" field to zero, and so gain access to resources reserved for members of the super user group. The kernel uses this field to protect against a process changing its user and group to become a super user (see the description of the **F$SUser** system call in the chapter on OS-9 System Calls).

The space at the end of the process descriptor for use as the system state stack (the stack used during system calls made by the process) is slightly more than 1k bytes. This is sufficient if all operating system components are written in assembly language. However, more and more device drivers and file managers are being written in C, which uses much more stack. Some file managers find it necessary to allocate a separate, larger system state stack for each process. The next two fields are available for management of such a stack. The current kernel (OS-9 version 2.4.3) does not use these fields:

Offset	Name	Size	Description
$3AC		b $454	System state stack. When the process is forked, its system state stack pointer is set to the top of this memory (the end of the process descriptor). This is therefore (automatically) the stack of the process when in system state (that is, when making a system call).

The high byte of the word **P$State** contains the bit flags that the kernel uses - in conjunction with the byte **P$QueuID** - to identify the current state of the process. The bit definitions for **P$State** (high byte) are:

Bit	Description when set
0	The process is dead. All its resources have been de-allocated except for the process descriptor itself, which will be de-allocated once the parent has received the process's exit status by executing a "wait for child" **F$Wait** system call, or the parent dies.
1	The process is condemned. When it next would start execution in user state it will be terminated.
4	The memory map permitted to the process has been altered. The SSM must build a new MMU memory map for the process before the process restarts execution in user state.
5	The time slice of the process has expired. When the process would next start execution in user state, the kernel will perform a reschedule of the active processes.
6	The process is in a timed sleep.
7	The process is executing a system call.

The possible values for **P$QueuID** (as ASCII characters) are:

Character	Description
	(space) no queue state - should not occur.
-	The process is not in a queue, and is not the current process (usually dead).
a	The process is in the active queue, waiting for processor time.
d	The process has been created for debugging, but is not yet in any queue, or has stopped at a breakpoint, or is executing without tracing ("at full speed").
e	The process is in an event queue, waiting on an event.
m	The process is waiting for a buffer - implemented by the **F$MBuf** system call of the Internet Support Package (ISP).
s	The process is in the sleep queue, in timed or untimed sleep.
w	The process is in the waiting queue, waiting for a child process to die.

Character	Description
*	The process is the current process. It is not in any queue.
&	The process is the system state debugger, in suspended state.

The field **P$frag** contains the root and end pointers to a linked list of structures used to keep track of unallocated memory fragments. Because the process could allocate memory of different colours, the list has the same structure as the system free memory colour node list. Each structure identifies the memory area from which it was taken, its colour and priority, and the first and last memory fragments from this area, as a linked list. Thus each process has a list of "locally free" memory areas that have been allocated to it from the free pool (and so are no longer in the free pool), but have not yet been allocated in response to a program request. The colour node structure is described in the section on memory allocation, in the chapter on OS-9 Modules, Memory, and Processes.

The field **P$Thread** contains the root and end pointers for the linked list of thread structures (or thread "blocks") allocated for this process. The end pointer points to the last structure in the linked list. Currently thread structures are only used for alarms (see the chapter on Inter-process Communication). Each structure represents a pending or cyclic alarm. Remaining thread structures are automatically de-allocated by the kernel when a process dies. The thread structure is shown below.

Offset	Name	Size	Description
$000	T_ID	w	Thread block type identifier. Because threads may be used for other purposes in the future, the alarm functions write a distinctive "magic number" here (45245) when allocating the thread block, and check it when deleting the alarm.
$002	T_Proc	w	Process ID of the process that created the block.
$004	T_MSiz	l	Size of memory allocated for the thread block - used when de-allocating the block.
$008	T_User	l	Group and user number of the process that created the block (the block owner). The System Process temporarily takes on this group and user number when executing the thread function.
$00C	T_Next	l	With **T_Prev**, the "next" and "previous" pointers linking this thread block into the linked list rooted in the System Globals (either **D_Thread** or **D_AlarTh**). Used by the System Process when checking for alarms that have reached their execution time. The linked list is maintained in execution time order.

$010	T_Prev	1	See **T_Next**, above.
$014	T_Link	1 2	"Next" and "previous" pointers respectively linking this thread block into the linked list rooted in the process descriptor of the creating process (see the field **P$Thread**).
$01C	T_Sys	1 4	Usage depends on thread block purpose. See below for the usage of this field with alarms.
$02C	T_Regs	1 18	Register stack frame, provided by the calling process for use when executing the thread function. For user state alarms this stack frame is automatically generated by the kernel, and provides the parameters and routine address for calling the **F$Send** system call.

The **T_Sys** field is structured as follows for alarm threads:

Offset	Name	Size	Description
$01C	T_Cycle	1	For cyclic alarms: number of ticks between alarms. Always zero for non-cyclic alarms.
$020	T_WkTime	1	For cyclic or relative alarms: the time for thread execution, as ticks since system startup (compared with **D_Ticks**). For absolute time alarms: time of day for thread execution, as seconds since midnight.
$024	T_WkDate	1	Only used for absolute time alarms - the Julian date for thread execution.