F$Alarm — Set Alarm Clock

ASM CALL: **OS9 F$Alarm**

INPUT: **d0.l = Alarm ID (or zero)**
d1.w = Alarm function code
d2.l = Signal code
d3.l = Time interval (or time)
d4.l = Date (when using absolute time)

OUTPUT: **d0.l = Alarm ID**

ERROR OUTPUT: **cc = carry bit set**
d1.w = error code if error

FUNCTION: F$Alarm creates an asynchronous software alarm clock timer. The timer sends a signal to the calling process when the specified time period has elapsed. A process may have multiple alarm requests pending.

The time interval is the number of system clock ticks (or 256ths of a second) to wait before an alarm signal is sent. If the high order bit is set, the low 31 bits are interpreted as 256ths of a second.

NOTE: All times are rounded up to the nearest clock tick.

The system automatically deletes a process's pending alarms when the process dies.

The alarm function code selects one of the several related alarm functions. Not all input parameters are always needed; each function is described in detail in the following pages.

OS-9 supports the following function codes:

A$Delete	**Remove a pending alarm request**
A$Set	**Send a signal after specified time interval**
A$Cycle	**Send a signal at specified time intervals**
A$AtDate	**Send a signal at Gregorian date/time**
A$AtJul	**Send a signal at Julian date/time**

SEE ALSO: F$Alarm System State Call

POSSIBLE ERRORS: E$UnkSvc, E$Param, E$MemFul, E$NoRAM, and E$BPAddr.

F$ALARM FUNCTION CODES:

A$Delete — Remove a Pending Alarm Request

INPUT: **d0.l = Alarm ID (or zero)**
d1.w = A$Delete function code

OUTPUT: **None**

ERROR OUTPUT: **cc = carry bit set**
d1.w = error code if error

FUNCTION: A$Delete removes a cyclic alarm, or any alarm that has not expired. If zero is passed as the alarm ID, all pending alarm requests are removed.

A$Set — Send a Signal after a Specified Time Interval

INPUT: **d0.l = Reserved, must be zero**
d1.w = A$Set function code
d2.w = Signal code
d3.l = Time Interval

OUTPUT: **d0.l = Alarm ID**

ERROR OUTPUT: **cc = carry bit set**
d1.w = error code if error

FUNCTION: A$Set sends one signal after the specified time interval has elapsed. The time interval may be specified in system clock ticks, or 256ths of a second.

A$Cycle — Send a Signal Every N Ticks/Seconds

INPUT: **d0.l = reserved, must be zero**
d1.w = A$Cycle function code
d2.l = signal code
d3.l = time interval (N)

OUTPUT: **d0.l = Alarm ID**

ERROR OUTPUT: **cc = carry bit set**
d1.w = error code if error

FUNCTION: A$Cycle is similar to the A$Set function, except that the alarm is reset after it is sent, to provide a recurring periodic signal.

A$AtDate — Send a Signal at Gregorian Date/Time

INPUT: **d0.l = Reserved, must be zero**
d1.w = A$AtDate function code
d2.l = Signal code
d3.l = Time (00hhmmss)
d4.l = Date (YYYYMMDD)

OUTPUT: **d0.l = Alarm ID**

ERROR OUTPUT: **cc = carry bit set**
d1.w = error code if error

FUNCTION: A$AtDate sends a signal to the caller at a specific date and time.

NOTE: A$AtDate only allows you to specify time to the nearest second. However, it does adjust if the system's date and time have changed (via F$STime). The alarm signal is sent anytime the system date/time becomes greater than or equal to the alarm time.

A$AtJul — Send a Signal at Julian Date/Time

INPUT: **d0.l = Reserved, must be zero**
d1.w = A$AtDate or A$AtJul function code
d2.l = Signal code
d3.l = Time (seconds after midnight)
d4.l = Date (Julian day number)

OUTPUT: **d0.l = Alarm ID**

ERROR OUTPUT: **cc = carry bit set**
d1.w = error code if error

FUNCTION: A$AtJul sends a signal to the caller at a specific Julian date and time. **NOTE:** A$AtJul only allows you to specify time to the nearest second. However, it does adjust if the system's date and time have changed (via F$STime). The alarm signal is sent anytime the system date/time becomes greater than or equal to the alarm time.

F$AllBit — Sends Bits in an Allocation Bit Map

ASM CALL: **OS9 F$AllBit**

INPUT: **d0.w = Bit number of first bit to set**
d1.w = Bit count (number of bits to set)
(a0) = Base address of an allocation bit map

OUTPUT: **None**

ERROR OUTPUT: **cc = carry bit set**
d1.w = error code if error

FUNCTION: F$AllBit sets bits in the allocation map that were found by F$SchBit, and are now allocated. Bit numbers range from 0 to n-1, where n is the number of bits in the allocation bit map.

In some applications you must allocate and deallocate segments of a fixed resource, such as memory. One convenient way is to set up a map that describes which blocks are available or in use. Each bit in the map represents one block. If the bit is set, the block is in use. If the bit is clear, the block is available. The F$SchBit, F$AllBit, and F$DelBit system calls perform the elementary bitmap operations of finding a free segment, allocating it, and returning it when it is no longer needed.

RBF uses these routines to manage cluster allocation on disks. They are accessible to users because they are occasionally useful.

SEE ALSO: F$SchBit and F$DelBit.

F$CCtl Cache Control

ASM CALL: **OS9 F$CCtl**

INPUT: **d0.l = desired cache control operation**

OUTPUT: **None**

ERROR OUTPUT: **cc = carry bit set**
d1.w = error code if error

FUNCTION: F$CCtl performs operations on the system instruction and/or data caches, if there are any.

If d0.l is set to zero, the system instruction and data caches are flushed. Non-super-group, user-state processes may perform this generic operation.

Only system-state processes (for example, device driver) and super-group processes may perform precise operation of F$CCtl. The following bits are defined in d0.l for precise operation:

Bit 0	If set, enables data cache.
Bit 1	If set, disables data cache.
Bit 2	If set, flushes data cache.
Bit 4	If set, enables instruction cache.
Bit 5	If set, disables instruction cache.
Bit 6	If set, flushes instruction cache.

All other bits are reserved. If any reserved bit is set, an E$Param error is returned.

Any program that builds or changes executable code in memory should flush the instruction cache by F$CCtl prior to the execution of the new code. This is necessary because the hardware instruction cache is not updated by data (write) accesses and may therefore contain the unchanged instruction(s). For example, if a subroutine builds an OS-9 system call on its stack, the F$CCtl system call to flush the instruction cache must execute prior to executing the temporary instructions.

POSSIBLE ERRORS: E$Param

F$Chain — Load and Execute New Primary Module

ASM CALL: **OS9 F$Chain**

INPUT: **d0.w = desired module type/language (must be program/object or 0=any)**
d1.l = additional memory size
d2.l = parameter size
d3.w = number of I/O paths to copy
d4.w = priority
(a0) = module name ptr
(a1) = parameter ptr

OUTPUT: **None: F$Chain does not return to the calling process.**

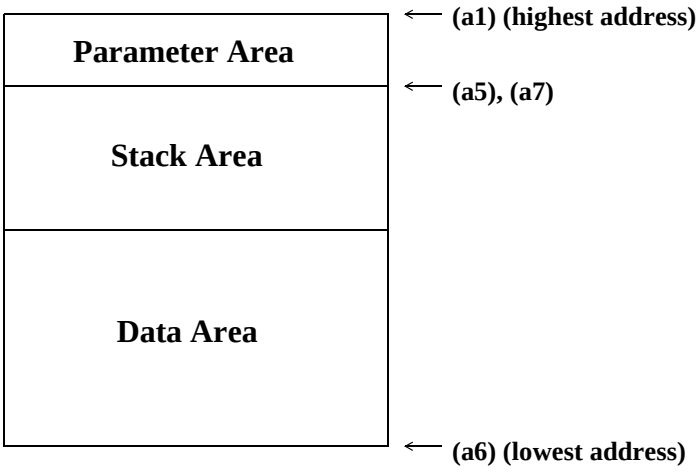
ERROR OUTPUT: **cc = carry bit set**
d1.w = error code if error

FUNCTION: F$Chain executes an entirely new program, but without the overhead of creating a new process. It is similar to a Fork command followed by an EXIT. F$Chain effectively resets the calling process's program and data memory areas and begins execution of a new primary module. Open paths are not closed or otherwise affected.

Chain executes as follows:

- ¿ The process's old primary module is unlinked.
- ¡ The system parses the name string of the new process's primary module (the program that will be executed). Next, the system module directory is searched to see if a module of the same name and type/language is already in memory. If so, the module is linked. If not, the name string is used as the pathlist of a file which is to be loaded into memory. The first module in this file is linked.
- ¬ The data memory area is reconfigured to the specified size in the new primary module's header.
- Ð Intercepts and any pending signals are erased.

The diagram below shows how Chain sets up the data memory area and registers for the new module (these are identical to F$Fork).



Registers passed to child process:

sr = 0000
pc = module entry point
d0.w = process ID
d1.l = group/user number
d2.w = priority
d3.w = number of I/O paths inherited
d4.l = undefined
d5.l = parameter size
d6.l = total initial memory allocation
d7.l = undefined

(a0) = undefined
(a1) = top of memory pointer
(a2) = undefined
(a3) = primary (forked) module pointer
(a4) = undefined
(a5) = parameter pointer
(a6) = static storage (data area) base pointer
(a7) = stack pointer (same as a5)

NOTE: (a6) is actually biased by $8000, but this can usually be ignored because the linker biases all data references by -$8000. However, it may be significant to note when debugging programs.

The minimum overall data area size is 256 bytes. Address registers point to even addresses.

SEE ALSO: F$Fork and F$Load.

CAVEATS: Most errors that occur during the Chain are returned as an exit status to the parent of the process doing the chain.

POSSIBLE ERRORS: E$NEMod

F$CmpNam — Compare Two Names

ASM CALL: **OS9 F$CmpNam**

INPUT: **d1.w = Length of pattern string**
(a0) = Pointer to pattern string
(a1) = Pointer to target string

OUTPUT: **cc = Carry bit clear if the strings match**

ERROR OUTPUT: **cc = carry bit set**
d1.w = error code if error

FUNCTION: F$CmpNam compares a target name to a source pattern to determine if they are equal. Upper and lower case are considered to match. Two wild card characters are recognized in the pattern string:

- Question mark (?) matches any single character
- Asterisk (*) matches any string

The target name must be terminated by a null byte.

POSSIBLE ERRORS:

E$Differ	The names do not match.
E$StkOvf	The pattern is too complex.

F$CpyMem — Copy External Memory

ASM CALL: **OS9 F$CpyMem**

INPUT: **d0.w = process ID of external memory's owner**
d1.l = number of bytes to copy
(a0) = address of memory in external process to copy
(a1) = caller's destination buffer pointer

OUTPUT: **None**

ERROR OUTPUT: **cc = carry bit set**
d1.w = error code if error

FUNCTION: F$CpyMem copies external memory into your buffer for inspection. You can use F$CpyMem to copy portions of the system's address space. This is especially helpful in examining modules. You can view any memory in the system with F$CpyMem.

SEE ALSO: F$Move

F$CRC — Generate CRC

ASM CALL: **OS9 F$CRC**

INPUT: **d0.l = Data byte count**
d1.l = CRC accumulator
(a0) = Pointer to data

OUTPUT: **d1.l = Updated CRC accumulator**

ERROR OUTPUT: **cc = carry bit set**
d1.w = error code if error

FUNCTION: F$CRC generates or checks the CRC (cyclic redundancy check) values of sections of memory. Compilers, assemblers, or other module generators use F$CRC to generate a valid module CRC.

If the CRC of a new module is to be generated, the CRC is accumulated over the entire module, excluding the CRC itself. The accumulated CRC is complemented and then stored in the correct position in the module.

You can calculate the CRC starting at the source address over a specified number of bytes. It is not necessary to cover an entire module in one call, since the CRC may be accumulated over several calls. The CRC accumulator must be initialized to $FFFFFFFF before the first F$CRC call for any particular module.

An easier method of checking an existing module's CRC is to perform the calculation on the entire module, including the module CRC. The CRC accumulator contains the CRC constant bytes if the module CRC is correct. The CRC constant is defined in sys.l and usr.l as CRCCon. Its value is $00800FE3.

SEE ALSO: ***OS-9 Technical Overview***, Chapter 1, section on CRC.

CAVEATS: The CRC value is three bytes long, in a four-byte field. To generate a valid module CRC, the caller must include the byte preceding the CRC in the check. This byte must be initialized to zero. For convenience, if a data pointer of zero is passed, the CRC is updated with one zero data byte. F$CRC always returns $FF in the most significant byte of d1, so d1.l may be directly stored (after complement) in the last four bytes of a module as the correct CRC.

F$DatMod — Create Data Module

ASM CALL: **OS9 F$DatMod**

INPUT: **d0.l = size of data required (not including header or CRC)**
d1.w = desired attr/revision
d2.w = desired access permission
d3.w = desired type/language (optional)
d4.l = memory color type (optional)
(a0) = module name string ptr

OUTPUT: **d0.w = module type/language**
d1.w = module attr/revision
(a0) = updated name string ptr
(a1) = module data ptr ('execution' entry)
(a2) = module header ptr

ERROR OUTPUT: **cc = carry bit set**
d1.w = error code if error

FUNCTION: F$DatMod creates a data module with the specified attribute/revision and clears the data portion of the module. The module is initially created with a valid CRC, and entered into the system module directory. Several processes can communicate with each other using a shared data module.

Be careful not to modify the data module's header or name string to avoid the possibility of the module becoming unknown to the system.

CAVEATS: The module created contains at least d0.l usable data bytes, but may be somewhat larger. The module itself will be larger by at least the size of the module header and CRC, and rounded up to the nearest system memory allocation boundary.

SEE ALSO: F$SetCRC and F$Move.

POSSIBLE ERRORS:

E$Differ	The names do not match.
E$StkOvf	The pattern is too complex.

F$DelBit — Deallocate in a Bit Map

ASM CALL: **OS9 F$DelBit**

INPUT: **d0.w = Bit number of first bit to clear**
d1.w = Bit count (number of bits to clear)
(a0) = Base address of an allocation bit map

OUTPUT: **None**

ERROR OUTPUT: **cc = carry bit set**
d1.w = error code if error

FUNCTION: F$DelBit clears bits in the allocation bit map that were previously allocated and are now free for general use. Bit numbers range from 0 to n-1, where n is the number of bits in the allocation bit map.

SEE ALSO: F$AllBitF$CpyMem and F$SchBit.

F$DExec Execute Debugged Program

ASM CALL: **OS9 F$DExec**

INPUT: **d0.w = process ID of child to execute**
d1.l = number of instructions to execute (0 = continuous)
d2.w = number of breakpoints in list
(a0) = breakpoint list
register buffer contains child register image

OUTPUT: **d0.l = total number of instructions executed so far**
d1.l = remaining count not executed
d2.w = exception occurred, if non-zero; exception offset
d3.w = classification word (addr or bus trap only)
d4.l = access address (addr or bus trap only)
d5.w = instruction register (addr or bus trap only)
register buffer updated

ERROR OUTPUT: **cc = carry bit set**
d1.w = error code if error

FUNCTION: F$DExec controls the execution of a suspended child process that has been created by the F$DFork call. The process performing F$DExec is suspended and its debugged child process is executed instead. Once the specified number of instructions are executed, a breakpoint is reached or an unexpected exception occurs, execution terminates, and control returns to the parent process. Thus, the parent and the child processes are never active at the same time.

F$DExec traces every instruction of the child process. It checks for the termination conditions after each instruction. Breakpoints are simply lists of addresses to check and work with ROMed object programs. Consequently, the child process being debugged runs at a slow speed.

If a -1 (hex $FFFFFFFF) is passed in d1.l, F$DExec replaces the instruction at each breakpoint address with an illegal opcode. It then executes the child process at full speed (with the trace bit clear) until a breakpoint is reached or the program terminates. This can save an enormous amount of time, but it is impossible for F$DExec to count the number of executed instructions.

Any OS-9 system calls made by the suspended program are executed at full speed and are considered one logical instruction. The same is true of system-state trap handlers. You cannot debug system-state processes.

The system uses the register buffer passed in the F$DFork call to save and restore the child's registers. Changing the contents of the register buffer alters the child process's registers.

If the child process terminates for any reason, the carry bit is set and returned. Tracing may continue as long as the child process does not perform a F$Exit (even after encountering any normally fatal error). A F$DExit call must be made to return the debugged process's resources (memory).

SEE ALSO: F$DFork and F$DExit.

CAVEATS: Tracing is allowed through user-state trap handlers, intercept routines, and the F$Chain system call. This is not a problem, but may seem strange at times.

POSSIBLE ERRORS: E$IPrcID and E$PrcAbt.

F$DExit Exit Debugged Program

ASM CALL: **OS9 F$DExit**

INPUT: **d0.w = process ID of child to terminate**

OUTPUT: **None**

ERROR OUTPUT: **cc = carry bit set**
d1.w = error code if error

FUNCTION: F$DExit terminates a suspended child process that was created with the F$DFork system call. To permit post-mortem examination, normal termination by the child process does not release any of its resources.

SEE ALSO: F$Exit, F$DFork, and F$DExec.

POSSIBLE ERRORS: E$IPrcID

F$DFork — Fork Process Under Control of Debugger

ASM CALL: **OS9 F$DFork**

INPUT: **d0.w = desired module type/revision (0 = any)**
d1.l = additional stack space to allocate (if any)
d2.l = parameter size
d3.w = number of I/O paths for child to inherit
d4.w = module priority
(a0) = module name ptr (or pathlist)
(a1) = parameter ptr
(a2) = register buffer: copy of child's (d0-d7/a0-a7/sr/pc)

OUTPUT: **d0.w = child process ID**
(a0) = updated past module name string
(a2) = initial image of the child process's registers in buffer

ERROR OUTPUT: **cc = carry bit set**
d1.w = error code if error

FUNCTION: F$DFork is similar to F$Fork, except that F$DFork creates a process whose execution can be closely controlled. The child process is not placed in the active queue but is left in a suspended state. This allows the debugger to control its execution through the special system calls F$DExec and F$DExit. (The child process is created with the trace bit of its status register set and is executed with the F$DExec system call.)

The register buffer is an area in the caller's data area that is permanently associated with each child process. It is set to an image of the child's initial registers for use with the F$DExec call.

For information about process creation, see the F$Fork service request.

SEE ALSO: F$DExit, F$Fork, and F$DExec.

CAVEATS: A process created by F$DFork does not execute unless it is told to do so. When a process is run, the trace bit is set in the user status register. This causes the system trace exception handler to occur once for each user instruction executed, thus user programs run slowly.

Processes whose primary module is owned by a super-user may only be debugged by a super-user. You cannot debug system-state processes.

F$Event Create, Manipulate, and Delete Events

ASM CALL: **OS9 F$Event**

INPUT: **d1.w = Event function code**
All others are dependent on function code

OUTPUT: **Dependent on function code**

ERROR OUTPUT: **Dependent on function code**

FUNCTION: Events are multiple-value semaphores that synchronize concurrent processes which share resources such as files, data modules, and CPU time. F$Event provides facilities to create and delete events, to permit processes to link/unlink events and obtain event information, to suspend operation until an event occurs, and for various means of signaling.

An OS-9 event is a 32-byte system global variable maintained by the system. The following fields are included in each event:

Event ID	This number and the event's array position are used to create a unique ID.
Event name	This name must be unique and cannot exceed 12 characters.
Event value	This four-byte integer value has a range of two billion.
Wait increment	This value is added to the event value when a process waits for the event. It is set when the event is created and does not change.
Signal increment	This value is added to the event value when the event is signaled. This value is set when the event is created and does not change.
Link Count	This is the event use count.
Next event	This is a pointer to the next process in the event queue. An event queue is circular and includes all processes waiting for the event. Each time the event is signaled, this queue is searched.
Previous event	This is a pointer to the previous process in the event queue.

The following function codes are supported:

Ev$Link	Link to existing event by name
Ev$UnLnk	Unlink event
Ev$Creat	Create new event
Ev$Delet	Delete existing event
Ev$Wait	Wait for event to occur
Ev$WaitR	Wait for relative to occur
Ev$Read	Read event value without waiting
Ev$Info	Return event information
Ev$Pulse	Signal an event occurrence
Ev$Signl	Signal an event occurrence
Ev$Set	Set event variable and signal an event occurrence
Ev$SetR	Set relative event variable; signal an event occurrence

POSSIBLE ERRORS: Dependent on function code

SEE ALSO: ***OS-9 Technical Overview*** Chapter 4, the section on Events.

F$EVENT FUNCTION CODES:

Ev$Link — Link to ExistingEvent by Name

INPUT: **(a0) = event name string pointer (max 11 chars)**
d1.w = 0 (Ev$Link function code)

OUTPUT: **d0.l = event ID number**
(a0) = updated past event name

ERROR OUTPUT: **cc = carry bit set**
d1.w = error code if error

FUNCTION: Ev$Link determines the ID number of an existing event. Once an event is linked, all subsequent references are made using the event ID returned. This permits the system to access events quickly, while protecting against programs using invalid or deleted events. The event use count is incremented when an Ev$Link is performed. To keep the use count synchronized properly, perform an Ev$UnLnk when the event will no longer be used.

POSSIBLE ERRORS:

E$BNam	Name is syntactically incorrect or longer than 11 chars.
E$EvNF	Event not found in the event table.

Ev$UnLnk — Unlink Event

INPUT: **d0.l = event ID number**
d1.w = 1 (Ev$UnLnk function code)

OUTPUT: **None**

ERROR OUTPUT: **cc = carry bit set**
d1.w = error code if error

FUNCTION: Ev$Unlnk informs the system that a process will no longer use an event. The event use count is decremented and the event is deleted when the count reaches zero. OS-9 uses this only for error checking.

POSSIBLE ERRORS:

E$EvntID	ID specified is not a valid active event.

Ev$Creat — Create New Event

INPUT: **d0.l = initial event variable value**
d1.w = 2 (Ev$Creat function code)
d2.w = auto-increment for Ev$Wait
d3.w = auto-increment for Ev$Signl
(a0) = event name string pointer (max 11-chars)

OUTPUT: **d0.l = event ID number**
(a0) = updated past event name

ERROR OUTPUT: **cc = carry bit set**
d1.w = error code if error

FUNCTION: Events may be created and deleted dynamically as needed. Upon creation, an initial signed value is specified, as well as signed increments to be applied each time the event occurs or is waited for. The event ID number returned is used in subsequent F$Event calls to refer to the event created.

POSSIBLE ERRORS:

E$BNam	Name is syntactically incorrect or longer than 11 characters.
E$EvFull	The event table is full.
E$EvBusy	The named event already exists.

Ev$Delet — Delete Existing Event

INPUT: **(a0) = event name string pointer (max 11-chars)**
d1.w = 3 (Ev$Delet function code)

OUTPUT: **(a0) = updated past event name**

ERROR OUTPUT: **cc = carry bit set**
d1.w = error code if error

FUNCTION: Ev$Delet removes an event from the system event table, freeing the entry for use by another event. Events have an implicit use count (initially set to one), which is incremented with each Ev$Link call and decremented with each Ev$UnLnk call. An event may not be deleted unless its use count is zero.

NOTE: OS-9 does not automatically unlink events when a F$Exit occurs.

POSSIBLE ERRORS:

E$BNam	Name is syntactically incorrect or longer than 11 characters.
E$EvNF	Event not found in the event table.
E$EvBusy	The event has a non-zero link count.

Ev$Wait — Wait for Event to Occur

INPUT: **d0.l = event ID number**
d1.w = 4 (Ev$Wait function code)
d2.l = minimum activation value (signed)
d3.l = maximum activation value (signed)

OUTPUT: **d1.l = actual event value**

ERROR OUTPUT: **cc = carry bit set**
d1.w = error code if error

FUNCTION: Ev$Wait waits for an event to occur. The event variable is compared to the range specified in d2 and d3. If the value is not in range, the calling process is suspended in a FIFO event queue. It waits until an Ev$Signl occurs that puts the value in range and adds the wait auto-increment (specified at creation) to the event variable.

If the process receives a signal while in the event queue, it is activated even though the event has not actually occurred. The auto-increment is not added to the event variable, and the event value returned is not within the specified range. The caller's intercept routine is executed, but an event error is not returned.

POSSIBLE ERRORS:

E$EvntID	ID specified is not a valid active event.

Ev$WaitR — Wait for Relative Event to Occur

INPUT: **d0.l = event ID number**
d1.w = 5 (Ev$WaitR function code)
d2.l = minimum relative activation value (signed)
d3.l = maximum relative activation value (signed)

OUTPUT: **d1.l = actual event value**
d2.l = minimum actual activation value
d3.l = maximum actual activation value

ERROR OUTPUT: **cc = carry bit set**
d1.w = error code if error

FUNCTION: Ev$WaitR works exactly like Ev$Wait, except that the range specified in d2 and d3 is relative to the current event value. The event value is added to d2 and d3 respectively, and the actual values are returned to the caller. The Ev$Wait function is then executed directly. If an underflow or overflow occurs on the addition, the values $80000000 (minimum integer), and $7fffffff (maximum integer) are used, respectively.

POSSIBLE ERRORS: E$EvntID ID specified is not a valid active event.

Ev$Read — Read Event Value Without Waiting

INPUT: **d0.l = event ID number**
d1.w = 6 (Ev$Read function code)

OUTPUT: **d1.l = current event value**

ERROR OUTPUT: **cc = carry bit set**
d1.w = error code if error

FUNCTION: Ev$Read reads the value of an event without waiting or modifying the event variable. You can use this to determine the availability of the event (or associated resource) without waiting.

POSSIBLE ERRORS: E$EvntID ID specified is not a valid active event.

Ev$Info — Return Event Information

INPUT: **d0.l = event index (ID number) to begin search**
d1.w = 7 (Ev$Info function code)
(a0) = ptr to buffer for event information

OUTPUT: **d0.l = event index found**
(a0) = data returned in buffer

ERROR OUTPUT: **cc = carry bit set**
d1.w = error code if error

FUNCTION: Ev$Info returns a copy of the 32-byte event table entry associated with an event. Unlike other F$Event functions, Ev$Info only uses the low word of d0. This index is the system event number, ranging from zero to the maximum number of system events minus one. The event information block for the first active event with an index greater than or equal to this index is returned in the caller's buffer. If none exists, an error is returned. Ev$Info is provided for utilities needing to determine the status of all active events.

POSSIBLE ERRORS:

E$EvntID	The index is above all active events.

Ev$Pulse — Signal an Event Occurrence

INPUT: **d0.l = event ID number**
d1.w = MS bit set to activate all processes in range
LS bits = 9 (Ev$Pulse function code)
d2.l = event pulse value

OUTPUT: **None**

ERROR OUTPUT: **cc = carry bit set**
d1.w = error code if error

FUNCTION: Ev$Pulse signals an event occurrence, but differs from Ev$Signl. The event variable is set to the value passed in d2, and the signal auto-increment is not applied. Then, the Ev$Signl search routine is executed and the original event value is restored.

POSSIBLE ERRORS:

E$EvntID	The ID specified is not a valid active event.

Ev$Signl — Signal an Event Occurrence

INPUT: **d0.l = event ID number**
d1.w = MS bit set to activate all processes in range
LS bits = 8 (Ev$Signl function code)

OUTPUT: **None**

ERROR OUTPUT: **cc = carry bit set**
d1.w = error code if error

FUNCTION: Ev$Signl signals that an event has occurred. The current event variable is updated with the signal auto-increment specified when the event was created. Then, the event queue is searched for the first process waiting for that event value. If the MS bit of d1 (the function code) is set, all processes in the event queue that have a value in range are activated. The sequence is the same for each event in the queue until the queue is exhausted:

- ¿ The signal auto-increment is added to the event variable.
- ¡ The first process in range is awakened.
- ¬ The event variable is updated with the wait auto-increment.
- Ð The search continues with the updated value.

POSSIBLE ERRORS:

E$EvntID	The ID specified is not a valid active event.

Ev$Set — Set Event Variable and Signal an Event Occurrence

INPUT: **d0.l = event ID number**
d1.w = MS bit set to activate all processes in range
LS bits = A (Ev$Set function code)
d2.l = new event value

OUTPUT: **d1.l = previous event value**

ERROR OUTPUT: **cc = carry bit set**
d1.w = error code if error

FUNCTION: Ev$Set is similar to the Ev$Signl call, except that the event variable is initially set to the value passed in d2 rather than updated with the signal auto-increment. After this is done, the Ev$Signl routine is executed directly.

POSSIBLE ERRORS:

E$EvntID	The ID specified is not a valid active event.

Ev$SetR — Set Relative Event Variable and Signal an Event Occurrence

INPUT: **d0.l = event ID number**
d1.w = MS bit set to activate all processes in range
LS bits = B (Ev$SetR function code)
d2.l = (signed) increment for event variable

OUTPUT: **d1.l = previous event value**

ERROR OUTPUT: **cc = carry bit set**
d1.w = error code if error

FUNCTION: Ev$SetR is similar to Ev$Signl, but instead of using the signal auto-increment value to update the event variable, the value in d2 is used. Arithmetic underflows or overflows are set to $80000000 or $7fffffff, respectively.

POSSIBLE ERRORS:

E$EvntID	The ID specified is not a valid active event.

F$Exit — Terminate the Calling Process

ASM CALL: **OS9 F$Exit**

INPUT: **d1.w = Status code to be returned to parent process**

OUTPUT: **Process is terminated**

ERROR OUTPUT: **cc = carry bit set**
d1.w = error code if error

FUNCTION: F$Exit is the means by which a process can terminate itself. Its data memory area is de-allocated and its primary module is unlinked. All open paths are automatically closed.

The death of the process can be detected by the parent executing a F$Wait call. This returns (to the parent) the status word passed by the child in its Exit call. The shell assumes that the status word is an OS-9 error code that the terminating process wishes to pass back to its parent process. The status word could also be a user-defined status value.

Processes called directly by the shell should only return an OS-9 error code or zero if no error occurred. **NOTE:** The parent *MUST* do a F$Wait before the process descriptor is returned.

A F$Exit call functions as follows:

- ¿ Close all paths.
- ¡ Return memory to system.
- ¬ Unlink primary module and user trap handlers.
- Ð Free process descriptor of any dead child processes.
- *f* If parent is dead, free the process descriptor.
- Ý If parent has not executed a F$Wait call, leave the process in limbo until parent notices the death.
- ý If parent is waiting, move parent to active queue, inform parent of death/status, remove child from sibling list, and free its process descriptor memory.

CAVEATS: Only the primary module and the user trap handlers are unlinked. Unlink any other modules that are loaded or linked by the process before calling F$Exit.

Although F$Exit closes any open paths, it pays no attention to errors returned by the F$Close request. Because of I/O buffering, this can cause write errors to go unnoticed when paths ARE left open. However, by convention, the standard I/O paths (0,1,2) are usually left open.

SEE ALSO: I$Close, F$SRtMem, F$UnLink, F$FindPD, F$RetPD, F$Fork, F$Wait, and F$AProc.

F$Fork — Create a New Process

ASM CALL: **OS9 F$Fork**

INPUT: **d0.w = desired module type/revision (usually program/object 0=any)**
d1.l = additional memory size
d2.l = parameter size
d3.w = number of I/O paths to copy
d4.w = priority
(a0) = module name pointer
(a1) = parameter pointer

OUTPUT: **d0.w = child process ID**
(a0) = updated beyond module name

ERROR OUTPUT: **cc = carry bit set**
d1.w = error code if error

FUNCTION: F$Fork creates a new process which becomes a child of the caller. It sets up the new process's memory, MPU registers, and standard I/O paths.

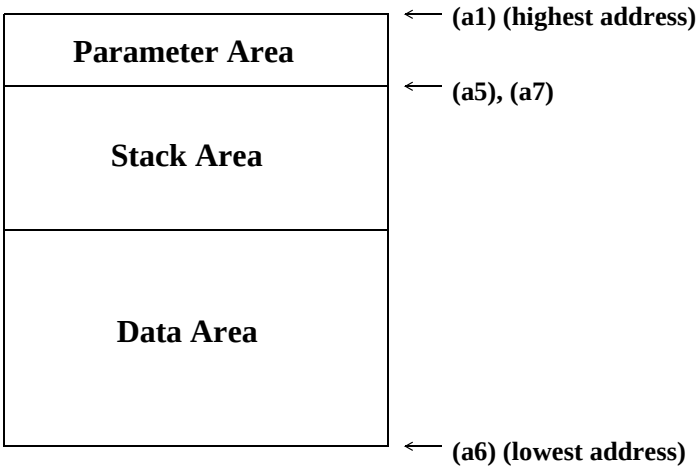
The system parses the name string of the new process's primary module (the program that will initially be executed). Next, the system module directory is searched to see if the program is already in memory. If so, the module is linked and executed. If not, the name string is used as the pathlist of the file which is to be loaded into memory. The first module in this file is linked and executed. To be loaded, the module must be program object code and have the appropriate read and/or execute permissions set for the user.

The primary module's module header is used to determine the process's initial data area size. OS-9 then attempts to allocate RAM equal to the required data storage size plus any additional size specified in d1, plus the size of any parameter passed. The RAM area must be contiguous.

The new process's registers are set up as shown in the diagram on the next page. The execution offset given in the module header is used to set the PC to the module's entry point. If d4.w is set to zero, the new process inherits the same priority as the calling process.

When the shell processes a command line, it passes a copy of the parameter portion (if any) of the command line as a parameter string. The shell appends an end-of-line character to the parameter string to simplify string-oriented processing.

If any of the these operations are unsuccessful, the fork is aborted and an error is returned to the caller. The diagram below shows how F$Fork sets up the data memory area and registers for a newly-created process. For more information, see F$Wait.



Registers passed to child process:

sr = 0000
pc = module entry point
d0.w = process ID
d1.l = group/user number
d2.w = priority
d3.w = number of I/O paths inherited
d4.l = undefined
d5.l = parameter size
d6.l = total initial memory allocation
d7.l = undefined

(a0) = undefined
(a1) = top of memory pointer
(a2) = undefined
(a3) = primary (forked) module pointer
(a4) = undefined
(a5) = parameter pointer
(a6) = static storage (data area) base pointer
(a7) = stack pointer (same as a5)

NOTE: (a6) will actually be biased by $8000, but this can usually be ignored because the linker biases all data references by -$8000. However, it may be significant to note when debugging programs.

CAVEATS: Both the child and parent process execute concurrently. If the parent executes a F$Wait call immediately after the fork, it waits until the child dies before it resumes execution. A child process descriptor is returned only when the parent does a F$Wait call.

Modules owned by a super-user execute in system state if the system-state bit in the module's attributes is set. This is rarely necessary, quite dangerous, and not recommended for beginners.

SEE ALSO: F$Wait, F$Exit, and F$Chain.

POSSIBLE ERRORS: E$IPrcID

F$GBlkMp Get Free Memory Block Map

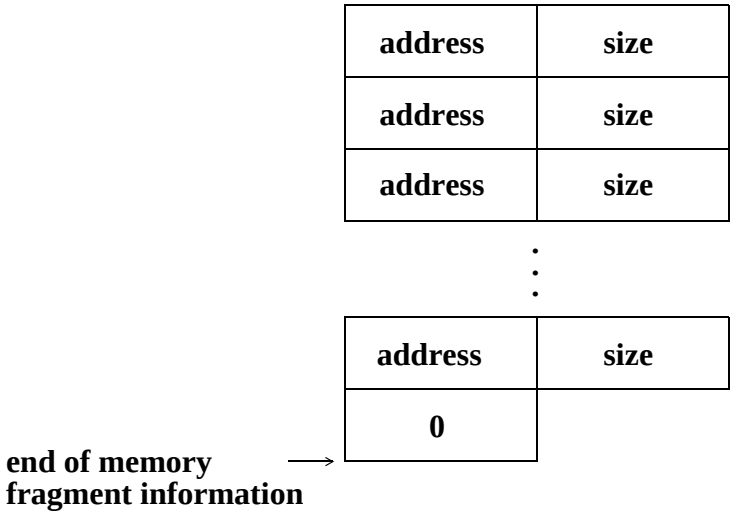
ASM CALL: **OS9 F$GblkMp**

INPUT: **d0.l = Address to begin reporting segments**
d1.l = Size of buffer in bytes
(a0) = Buffer pointer

OUTPUT: **d0.l = System's minimum memory allocation size**
d1.l = Number of memory fragments in system
d2.l = Total RAM found by system at startup
d3.l = Current total free RAM available
(a0) = Memory fragment information

ERROR OUTPUT: **cc = carry bit set**
d1.w = error code if error

FUNCTION: F$GBlkMp copies the address and size of the system's free RAM blocks into the user's buffer for inspection. It also returns various information concerning the free RAM as noted by the output registers above. The address and size of the free RAM blocks are returned in the user's buffer in following format (address and size are 4-bytes):



Although F$GblkMp returns the address and size of the system's free memory blocks, these blocks may never be accessed directly. Use F$SRqMem to request free memory blocks.

SEE ALSO: F$SRqMem and F$Mem.

CAVEATS: F$GBlkMp provides a status report concerning free system memory for mfree and similar utilities. The address and size of free RAM changes with system use. Although F$GblkMp returns the address and size of the system's free memory blocks, these blocks may never be accessed directly. Use F$SRqMem to request free memory blocks.

F$GModDr Get Copy of Module Directory

ASM CALL: **OS9 F$GModDr**

INPUT: **d1.l = Maximum number of bytes to copy**
(a0) = Buffer pointer

OUTPUT: **d1.l = Actual number of bytes copied**

ERROR OUTPUT: **cc = Carry bit set**
d1.w = Appropriate error code

FUNCTION: F$GModDr copies the system's module directory into the user's buffer for inspection. mdir uses F$GModDr to look at the module directory. Although the module directory contains pointers to each module in the system, the modules should never be accessed directly. Rather, use a F$CpyMem call to copy portions of the system's address space for inspection. On some systems, directly accessing the modules may cause address or bus trap errors.

SEE ALSO: F$Move and F$CpyMem.

CAVEATS: This system call is provided primarily for use by mdir and similar utilities. The format and contents of the module directory may change on different releases of OS-9. For this reason, it is often preferable to use the output of mdir to determine the names of modules in memory.

F$GPrDBT Get Copy of Process Descriptor Block Table

ASM CALL: **OS9 F$GPrDBT**

INPUT: **d1.l = maximum number of bytes to copy**
(a0) = Buffer pointer

OUTPUT: **d1.l = Actual number of bytes copied**

ERROR OUTPUT: **cc = Carry bit set**
d1.w = Appropriate error code

FUNCTION: F$GPrDBT copies the process descriptor block table into the caller's buffer for inspection. The procs utility uses F$GPrDBT to quickly determine which processes are active in the system. Although F$GPrDBT returns pointers to the process descriptors of all processes, *NEVER* access the process descriptors directly. Instead, use the F$GPrDsc system call if you need to inspect particular process descriptors.

The system call, F$AllPd, describes the format of the process descriptor block table.

SEE ALSO: F$GPrDsc and F$AllPd.

F$GPrDsc Get Copy of the Process Descriptor

ASM CALL: **OS9 F$GPrDsc**

INPUT: **d0.w = Requested process ID**
d1.w = Number of bytes to copy
(a0) = Process descriptor buffer pointer

OUTPUT: **None**

ERROR OUTPUT: **cc = Carry bit set**
d1.w = Appropriate error code

FUNCTION: F$GPrDsc copies a process descriptor into the caller's buffer for inspection. There is no way to change data in a process descriptor. The procs utility uses F$GPrDsc to gain information about an existing process.

SEE ALSO: F$GPrDBT

CAVEATS: The format and contents of a process descriptor may change with different releases of OS-9.

POSSIBLE ERRORS: E$PrcID

F$Gregor — Get Gregorian Date

ASM CALL: **OS9 F$Gregor**

INPUT: **d0.l = time (seconds since midnight)**
d1.l = Julian date

OUTPUT: **d0.l = time (00hhmmss)**
d1.l = date (yyyymmdd)

ERROR OUTPUT: **cc = Carry bit set**
d1.w = Appropriate error code

FUNCTION: F$Gregor converts Julian dates to Gregorian dates. Gregorian dates are considered the normal calendar dates.

The Julian date is similar to the Julian date used by astronomers. It is based on the number of days that have elapsed since January 1, 4713 B.C. Each astronomical Julian day changes at noon. OS-9 differs slightly from the astronomical standard by changing Julian dates at midnight. It is relatively easy to adjust for this, when necessary.

CAVEATS: The normal (Gregorian) calendar was revised to correct errors due to leap year at different dates throughout the world. The algorithm used by OS-9 makes this adjustment on October 15, 1582. Be careful when you are working with old dates, because the same day may be recorded as a different date by different sources.

NOTE: F$Gregor is the inverse function of F$Julian.

SEE ALSO: F$Julian and F$Time.

F$ID Get Process ID / User ID

ASM CALL: **OS9 F$ID**

INPUT: **None**

OUTPUT: **d0.w = Current process ID**
d1.l = Current process group/user number
d2.w = Current process priority

ERROR OUTPUT: **cc = Carry bit set**
d1.w = Appropriate error code

FUNCTION: F$ID returns the caller's process ID number, group and user ID, and current process priority (all word values). The process ID is assigned by OS-9 and is unique to the process. The user ID is defined in the system password file, and is used for system and file security. Several processes can have the same user ID.

F$Icpt — Set Up a Signal Intercept Trap

ASM CALL: **OS9 F$Icpt**

INPUT: **(a0)** = **Address of the intercept routine**
(a6) = **Address to be passed to the intercept routine**

OUTPUT: **Signals sent to the process will cause the intercept routine to be called instead of the process being killed.**

ERROR OUTPUT: **None**

FUNCTION: F$Icpt tells OS-9 to install a signal intercept routine: (a0) contains the address of the signal handler routine, and (a6) usually contains the address of the program's data area.

After the F$Icpt call has been made, whenever the process receives a signal, its intercept routine executes. A signal aborts a process which has not used the F$Icpt service request and its termination status (register d1.w) is the signal code. Many interactive programs set up an intercept routine to handle keyboard abort and keyboard interrupt signals.

The intercept routine is entered asynchronously because a signal may be sent at any time (similar to an interrupt) and is passed the following:

d1.w = **Signal code**
(a6) = **Address of intercept routine data area**

The intercept routine should be short and fast, such as setting a flag in the process's data area. Avoid complicated system calls (such as I/O). After the intercept routine is complete, it may return to normal process execution by executing the F$RTE system call.

SEE ALSO: F$RTE and F$Send.

CAVEATS: Each time the intercept routine is called, 70 bytes are used on the user's stack.

F$Julian Get Julian Date

ASM CALL: **OS9 F$Julian**

INPUT: **d0.l = time (00hhmmss)**
d1.l = date (yyyymmdd)

OUTPUT: **d0.l = time (seconds since midnight)**
d1.l = Julian date

ERROR OUTPUT: **cc = Carry bit set**
d1.w = Appropriate error code

FUNCTION: F$Julian converts Gregorian dates to Julian dates.

Julian dates are very convenient for computing elapsed time. To compute the number of days between two dates, subtract the lower Julian date number from the higher number.

The Julian day number returned is similar to the Julian date used by astronomers. It is based on the number of days that have elapsed since January 1, 4713 B.C. Each astronomical Julian day changes at noon. OS-9 differs slightly from the astronomical standard by changing Julian dates at midnight. It is relatively easy to adjust for this, when necessary.

You can also use the Julian day number to determine the day of the week for a given date. Use the following formula:

weekday = MOD(Julian_Date + 2, 7)

This returns the day of the week as 0 = Sunday, 1 = Monday, etc.

CAVEATS: The normal (Gregorian) calendar was revised to correct errors due to leap year at different dates throughout the world. The algorithm used by OS-9 makes this adjustment on October 15, 1582. Be careful when working with old dates, because the same day may be recorded as a different date by different sources.

F$Link — Link to Memory Module

ASM CALL: **OS9 F$Link**

INPUT: **d0.w = Desired module type/language byte (0 = any)**
(a0) = Module name string pointer

OUTPUT: **d0.w = Actual module type/language**
d1.w = Module attributes/revision level
(a0) = Updated past the module name
(a1) = Module execution entry point
(a2) = Module pointer

ERROR OUTPUT: **cc = Carry bit set**
d1.w = Appropriate error code

FUNCTION: F$Link causes OS-9 to search the module directory for a module having a name, language, and type as given in the parameters. If found, the address of the module's header is returned in (a2). The absolute address of the module's execution entry point is returned in (a1). As a convenience, you can obtain this and other information from the module header. The module's link count is incremented to keep track of how many processes are using the module. If the module requested is not re-entrant, only one process may link to it at a time.

If the module's access word does not give the process read permission, the link call fails. Link also fails to find modules whose header has been destroyed (altered or corrupted) in memory.

SEE ALSO: F$Load, F$UnLink, and F$UnLoad.

POSSIBLE ERRORS: EMNF, EBNam, and E$ModBsy.

F$Load — Load Module(s) from a File

ASM CALL: **OS9 F$Load**

INPUT: **d0.b = Access mode**
d1.l = Memory "color" type to load (optional)
(a0) = Pathname string pointer

OUTPUT: **d0.w = Actual module type/language**
d1.w = Attributes/revision level
(a0) = Updated beyond path name
(a1) = Module execution entry pointer (of first module loaded)
(a2) = Module pointer

ERROR OUTPUT: **cc = Carry bit set**
d1.w = Appropriate error code

FUNCTION: F$Load opens a file specified by the pathlist. It reads one or more memory modules from the file into memory until it reaches an error or end of file. Then, it closes the file. Modules are usually loaded into the highest physical memory available.

An error can indicate an actual I/O error, a module with a bad parity or CRC, or that the system memory is full.

All modules that are loaded are added to the system module directory, and the first module read is linked. The parameters returned are the same as those returned by a link call, and apply only to the first module loaded.

To be loaded, the file must contain a module or modules that have a proper module header and CRC. The access mode may be specified as either Exec_ or Read_, causing the file to load from the current execution or data directory, respectively.

If any of the modules loaded belong to the super-user, the file must also be owned by the super-user. This prevents normal users from executing privileged service requests.

The input register which specifies memory color type (d1.l) is only referenced if the most significant bit of d0.b is set.

CAVEATS: F$Load does not work on SCF devices.

POSSIBLE ERRORS: E$MemFul and E$BMID.

F$Mem — Resize Data Memory Area

ASM CALL: **OS9 F$Mem**

INPUT: **d0.l = Desired new memory size in bytes**

OUTPUT: **d0.l = Actual size of new memory area in bytes**
(a1) = Pointer to new end of data segment (+1)

ERROR OUTPUT: **cc = Carry bit set**
d1.w = Appropriate error code

FUNCTION: F$Mem contracts or expands the process's data memory area. The new size requested is rounded up to an even memory allocation block (16 bytes in version 2.0). Additional memory is allocated contiguously upward (towards higher addresses), or de-allocated downward from the old highest address. If d0 equals zero, the call is considered an information request and the current upper bound and size is returned.

This request can never return all of a process's memory, or cause deallocation of memory at its current stack pointer.

The request may return an error upon an expansion request even though adequate free memory exists, because the data area must always be contiguous. Memory requests by other processes may fragment memory into smaller, scattered blocks that are not adjacent to the caller's present data area.

POSSIBLE ERRORS: E$DelSP, E$MemFul, and E$NoRAM.

F$PErr Print Error Message

ASM CALL: **OS9 F$PErr**

INPUT: **d0.w = Error message path number (0=none)**
d1.w = Error number

OUTPUT: **None**

ERROR OUTPUT: **None**

FUNCTION: F$PErr is the system's error reporting facility. It writes an error message to the standard error path. Most OS-9 systems will print ERROR #mmm.nnn. Error numbers 000:000 to 063:255 are reserved for the operating system.

If an error path number is specified, the path is searched for a text description of the error encountered. The error message path contains an ASCII file of error messages. Each line may be up to 80 characters long. If the error number matches the first seven characters in a line (that is, 000:215), the rest of the line is printed along with the error number.

Error messages may be continued on several lines by beginning each continuation line with a space. An example error file might contain lines like this:

000:214 (E$FNA) File not accessible.

An attempt to open a file failed. The file was found, but is inaccessible to you in the requested mode. Check the file's owner ID and access attributes.

000:215 (E$BPNam) Bad pathlist specified.

The pathlist specified is syntactically incorrect.

000:216 (E$PNNF) File not found.

The pathlist does not lead to any known file.

000:218 (E$CEF) Tried to create a file that already exists.

000:253 (E$Share) Non-sharable file busy.

The most common way to get this error is to try to delete a file that is currently open. Anytime a file already in use is opened for non-sharable access, this error occurs. It also occurs if you try to access a non-sharable device (for example, a printer) that is busy.

F$PrsNam — Parse a Path Name

ASM CALL: **OS9 F$PrsNam**

INPUT: **(a0) = Name of string pointer**

OUTPUT: **d0.b = Pathlist delimiter**
d1.w = Length of pathlist element
(a0) = Pathlist pointer updated past the optional "/" character
(a1) = Address of the last character of the name +1

ERROR OUTPUT: **cc = Carry bit set**
d1.w = Appropriate error code

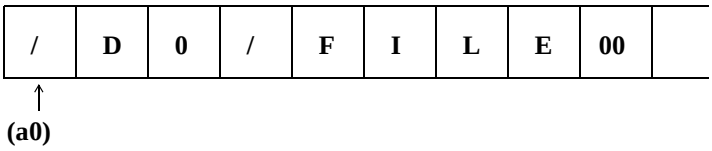
FUNCTION: F$PrsNam parses a string for a valid pathlist element, returning its size. Note that this does not parse an entire pathname, only one element in it. A valid pathlist element may contain the following characters:

A - Z	**Upper case letters**	**.**	**Periods**
a - z	**Lower case letters**	**_**	**Underscores**
0 - 9	**Numbers**	**$**	**Dollar signs**

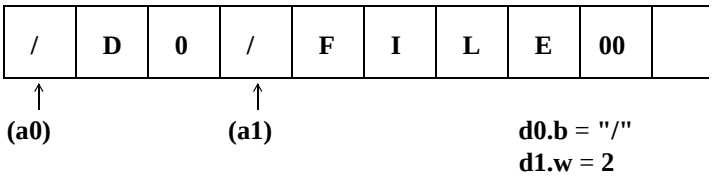
Any other character terminates the name and is returned as the pathlist delimiter.

NOTE: F$PrsNam processes only one name, so several calls may be needed to process a pathlist that has more than one name. F$PrsNam terminates a name on recognizing a delimiter character. Pathlists are usually terminated with a null byte.

BEFORE F$PrsNam CALL:



AFTER F$PrsNam CALL:



SEE ALSO: F$CmpNam

POSSIBLE ERRORS: E$BNam

F$RTE Return from Interrupt Exception

ASM CALL: **OS9 F$RTE**

INPUT: **None**

OUTPUT: **None**

FUNCTION: F$RTE may be used to exit from a signal processing routine.

F$RTE terminates a process signal intercept routine and continues execution of the main program. However, if there are unprocessed signals pending, the interrupt routine executes again (until the queue is exhausted) before returning to the main program.

CAVEATS: When a signal is received, 70 bytes are used on the user stack. Consequently, intercept routines should be kept very short and fast if many signals are expected.

SEE ALSO: F$Icpt

F$SchBit — Search Bit Map for a Free Area

ASM CALL: **OS9 F$SchBit**

INPUT: **d0.w = Beginning bit number to search**
d1.w = Number of bits needed
(a0) = Bit map pointer
(a1) = End of bit map (+1) pointer

OUTPUT: **d0.w = Beginning bit number found**
d1.w = Number of bits found

ERROR OUTPUT: **cc = Carry bit set**
d1.w = Appropriate error code

FUNCTION: F$SchBit searches the specified allocation bit map for a free block (cleared bits) of the required length, starting at the beginning bit number (d0.w). F$SchBit returns the offset of the first block found of the specified length.

If no block of the specified size exists, it returns with the carry set, beginning bit number, and size of the largest block found.

SEE ALSO: F$AllBit and F$DelBit.

F$Send — Send a Signal to Another Process

ASM CALL: **OS9 F$Send**

INPUT: **d0.w = Intended receiver's process ID number (0 = all)**
d1.w = Signal code to send

OUTPUT: **None**

ERROR OUTPUT: **cc = Carry bit set**
d1.w = Appropriate error code

FUNCTION: F$Send sends a signal to a specific process. The signal code is a word value. A process may send the same signal to multiple processes of the same Group/User ID by passing 0 as the receiver's process ID number. For example, the OS-9 Shell command, kill 0, will unconditionally abort all processes with the same group/user ID (except the Shell itself). This is a handy but dangerous tool to get rid of unwanted background tasks.

If you attempt to send a signal to a process that has an unprocessed, previous signal pending, the signal is placed in a FIFO queue of signals for the individual process. If the process is in the signal intercept routine when it receives a signal, the new signal is processed when F$RTE executes.

If the destination process for the signal is sleeping or waiting, it is activated so that it may process the signal. The signal processing intercept routine is executed, if it exists (see F$Icpt), otherwise the signal aborts the destination process, and the signal code becomes the exit status (see F$Wait).

An exception is the wakeup signal. It activates a sleeping process but does not cause examination of the signal intercept routine and will not abort a process that has not made an F$Icpt call.

Some of the signal codes have meanings defined by convention:

S$Kill = 0 = System abort (unconditional)
S$Wake = 1 = Wake up process
S$Abort = 2 = Keyboard abort
S$Intrpt = 3 = Keyboard interrupt
S$HangUp = 4 = Modem Hangup
5-31 = Reserved for Microware; deadly to I/O
32-255 = Reserved for Microware
256-65535 = User defined

The S$Kill signal may only be sent to processes with the same group ID as the sender. Super users may kill any process.

CAVEAT: The I/O system uses the S$Wake signal extensively. It is not reliable if used by user-state programs.

Signal values less than 32 (S$Deadly) usually cause the current I/O operation to terminate with an error status equal to the signal value.

SEE ALSO: F$Wait, F$Icpt, and F$Sleep.

POSSIBLE ERRORS: E$IPrcID and E$USigP.

F$SetCRC — Generate Valid CRC in Module

ASM CALL: **OS9 F$SetCRC**

INPUT: **(a0) = module pointer**

OUTPUT: **None**

ERROR OUTPUT: **cc = Carry bit set**
d1.w = Appropriate error code

FUNCTION: F$SetCRC updates the header parity and CRC of a module in memory. The module may be an existing module known to the system, or simply an image of a module that will subsequently be written to a file. The module must have correct size and sync bytes; other parts of the module are not checked.

SEE ALSO: F$CRC

CAVEATS: The module image must start on an even address or an address error occurs.

OS-9 does not permit any modification to the header of a module known to the system. Modifying the header makes the module inaccessible to other processes.

POSSIBLE ERRORS: E$BMID

F$SetSys — Set/Examine OS-9 System Global Variables

ASM CALL: **OS9 F$SetSys**

INPUT: **d0.w = offset of system global variable to set/examine**
d1.l = size of variable in least significant word (1, 2 or 4 bytes).
The most significant bit, if set, indicates an examination request. Otherwise, the variable is changed to the value in register d2.
d2.l = new value (if change request)

OUTPUT: **d2.l = original value of system global variable**

ERROR OUTPUT: **cc = Carry set**
d1.w = Appropriate error code

FUNCTION: F$SetSys changes or examines a system global variable. These variables have a D_ prefix in the system library sys.l. Consult the DEFS files for a description of the system global variables.

SEE ALSO: F$SPrior and the DEFS Files section in the ***OS-9 Technical I/O Manual***

CAVEATS: Only a super-user can change system variables. Any system variable may be examined, but only a few may be altered. The only useful variables that may be changed are D_MinPty and D_MaxAge. Consult Chapter 2 (section on process scheduling) of the ***OS-9 Technical Overview*** for an explanation of what these variables control.

The system global variables are OS-9's data area. It is highly likely that they will change from one release to another. You will probably have to relink programs using this system call to run them on future versions of OS-9.

CAUTION: The super-user must be extremely careful when changing system global variables.

F$Sigmask Masks/Unmasks Signals During Critical Code

ASM CALL: **OS9 F$SigMask**

INPUT: **d0.l = reserved, must be zero**
d1.l = process signal level
0 = clear
1 = set/increment
-1 = decrement

OUTPUT: **none**

ERROR OUTPUT: **cc = carry bit set**
d1.w = error code if error

FUNCTION: F$SigMask enables or disables signals from reaching the calling process. If a signal is sent to a process whose mask is disabled, the signal is queued until the process mask becomes enabled. The process's signal intercept routine is executed with signals inherently masked.

Two exceptions to this rule are the S$Kill and S$Wake signals. S$Kill terminates the receiving process, regardless of the state of its mask. S$Wake ensures that the process is active, but does not queue.

When a process makes a F$Sleep or F$Wait system call, its signal mask is automatically cleared. If a signal is already queued, these calls return immediately (to the intercept routine).

NOTE: Signals are analogous to hardware interrupts. They should be masked sparingly, and intercept routines should be as short and fast as possible.

F$Sleep — Put Calling Process to Sleep

ASM CALL: **OS9 F$Sleep**

INPUT: **d0.l = Ticks/seconds (length of time to sleep)**

OUTPUT: **d0.l = Remaining number of ticks if awakened prematurely**

ERROR OUTPUT: **cc = Carry bit set**
d1.w = Appropriate error code

FUNCTION: F$Sleep deactivates the calling process until the number of ticks requested have elapsed. Sleep(0) sleeps indefinitely. Sleep(1) gives up a time slice but does not necessarily sleep for one tick. You cannot use F$Sleep to time more accurately than + or - 1 tick, because it is not known when the F$Sleep request was made during the current tick.

A sleep of one tick is effectively a "give up current time slice" request; the process is immediately inserted into the active process queue and resumes execution when it reaches the front of the queue.

A sleep of two or more (n) ticks causes the process to be inserted into the active process queue after (n - 1) ticks occur and resumes execution when it reaches the front of the queue. The process is activated before the full time interval if a signal (in particular S$Wake) is received. Sleeping indefinitely is a good way to wait for a signal or interrupt without wasting CPU time.

The duration of a tick is system dependent, but is usually .01 seconds. If the high order bit of d0.l is set, the low 31 bits are converted from 256ths of a second into ticks before sleeping to allow program delays to be independent of the system's clock rate.

SEE ALSO: F$Send and F$Wait.

CAVEATS: The system clock must be running to perform a timed sleep. The system clock is not required to perform an indefinite sleep or to give up a time-slice.

POSSIBLE ERRORS: E$NoClk

F$SPrior Set Process Priority

ASM CALL: **OS9 F$SPrior**

INPUT: **d0.w = Process ID number**
d1.w = Desired process priority: 65535 = highest
0 = lowest

OUTPUT: **None**

ERROR OUTPUT: **cc = Carry bit set**
d1.w = Appropriate error code

FUNCTION: F$SPrior changes the process priority to the new value specified. A process can only change another process's priority if it has the same user ID. The one exception to this rule is a super user (group ID zero), which may alter any process's priority.

There are two system global variables that affect task-switching. D_MinPty is the minimum priority that a task must have for OS-9 to age or execute it. D_MaxAge is the cutoff aging point. D_MinPty and D_MaxAge are initially set in the Init module.

SEE ALSO: F$SetSys and the section on process scheduling in Chapter 2 of the ***OS-9 Technical Overview.***

CAVEATS: A very small change in relative priorities has a large effect. For example, if two processes have priorities 100 and 200, the process with the higher priority runs 100 times before the low priority process runs at all. In actual practice, the difference may not be this extreme because programs spend a lot of time waiting for I/O devices.

POSSIBLE ERRORS: E$IPrcID

F$SRqCMem — System Request for Colored Memory

ASM CALL: **OS9 F$SRqCMem**

INPUT: **d0.l = Byte count of requested memory**
d1.l = Memory type code (0 = any)

OUTPUT: **d0.l = Byte count of memory granted**
(a2) = Pointer to memory block allocated

ERROR OUTPUT: **cc = Carry bit set**
d1.w = Appropriate error code

FUNCTION: F$SRqCMem allocates a block of a specific type of memory. If a non-zero type is requested, the search is restricted to memory areas of that type. The area with the highest priority is searched first.

When the type code is zero, the search is based only on priority. This allows you to configure a system so that fast on-board memory is allocated before slow off-board memory. Areas with a priority of zero are excluded from the search.

If more than one memory area has the same priority, the area with the largest total free space is searched first. This allows memory areas to be balanced (that is, contain approximately equal amounts of free space).

Memory types or "color codes" are system dependent and may be arbitrarily assigned by the system administrator. Values below 256 are reserved for Microware use.

The number of bytes requested are rounded up to a system defined blocksize, which is currently 16 bytes. The memory always begins on an even boundary.

If -1 is passed in d0.l, the largest block of free memory of the specified type is allocated to the calling process.

F$SRqMem is equivalent to a F$SRqCMem request with a color of zero.

SEE ALSO: F$SRqMem, F$SRtMem, and F$Mem; Init module memory definitions and Colored Memory discussion in Chapter 2 of the ***OS-9 Technical Overview.***

POSSIBLE ERRORS: E$MemFul, E$NoRAM, and E$Damage.

F$SrqMem System Memory Request

ASM CALL: **OS9 F$SRqMem**

INPUT: **d0.l = Byte count of requested memory**

OUTPUT: **d0.l = Byte count of memory granted**
(a2) = Pointer to memory block allocated

ERROR OUTPUT: **cc = Carry bit set**
d1.w = Appropriate error code

FUNCTION: F$SRqMem allocates a block of memory from the top of available RAM. The number of bytes requested is rounded up to a system defined blocksize (currently 16 bytes). This system call is useful for allocating I/O buffers and any other semi-permanent memory. The memory always begins on an even boundary.

If -1 is passed in d0.l, the largest block of free memory is allocated to the calling process.

The maximum number of blocks any process may have allocated is 32. This includes the primary module's static storage area. **NOTE:** This is a limit on the number of segments allocated, not the amount of memory.

SEE ALSO: F$SRtMem and F$Mem.

CAVEATS: The byte count of memory allocated (as well as the pointer to the block allocated) must be saved if the memory is ever to be returned to the system.

POSSIBLE ERRORS: E$MemFul and E$NoRAM.

F$SRtMem — Return System Memory

ASM CALL: **OS9 F$SRtMem**

INPUT: **d0.l = Byte count of memory being returned**
(a2) = Address of memory block being returned

OUTPUT: **None**

ERROR OUTPUT: **cc = Carry bit set**
d1.w = Appropriate error code

FUNCTION: F$SRtMem de-allocates memory after it is no longer needed. The number of bytes returned is rounded up to a system defined blocksize before the memory is returned. Rounding occurs identically to that done by F$SRqMem.

In user state, the system keeps track of memory allocated to a process and all blocks not returned are automatically de-allocated by the system when a process terminates. In system state, the process must explicitly return its memory.

SEE ALSO: F$SRqMem and F$Mem.

POSSIBLE ERRORS: E$BPAddr

F$SSpd Suspend Process

ASM CALL: **OS9 F$SSpd**

INPUT: **d0.w = process ID to suspend**

OUTPUT: **None**

ERROR OUTPUT: **cc = Carry bit set**
d1.w = Appropriate error code

FUNCTION: F$SSpd is currently not implemented.

SEE ALSO: F$SetPri and F$SetSys.

CAVEATS: You can suspend a process by setting its priority below the system's minimum executable priority level (D_SysMin).

F$STime — Set System Date and Time

ASM CALL: **OS9 F$STime**

INPUT: **d0.l = current time (00hhmmss)**
d1.l = current date (yyyymmdd)

OUTPUT: **Time/date is set**

ERROR OUTPUT: **cc = Carry bit set**
d1.w = Appropriate error code

FUNCTION: F$STime sets the current system date/time and starts the system real-time clock to produce time-slice interrupts. F$STime is accomplished by putting the date/time packet in the system direct storage area, and then linking the clock module. The clock initialization routine is called if the link is successful.

It is the function of the clock module to:

- ¿ Set up any hardware dependent functions to produce system tick interrupts (including moving new date/time into hardware, if needed).
- ¡ Install a service routine to clear the interrupt when a tick occurs.

The OS-9 kernel keeps track of the current date and time in software to make clock modules small and simple. Certain utilities and functions in OS-9 expect the clock to be running with an accurate date and time. For this reason, always run F$STime when the system is started. This is usually done in the system startup file.

SEE ALSO: F$Link and F$Time.

CAVEATS: The date and time are not checked for validity. On systems with a battery-backed clock, it is usually only necessary to supply the year to the F$STime call. The actual date and time are read from the real-time clock.

F$STrap Set Error Trap Handler

ASM CALL: **OS9 F$STrap**

INPUT: **(a0) = Stack to use if exception occurs**
(or zero to use the current stack)
(a1) = Pointer to service request initialization table

OUTPUT: **None**

ERROR OUTPUT: **cc = Carry bit set**
d1.w = Appropriate error code

FUNCTION: F$STrap enters process local Error Trap routine(s) into the process descriptor dispatch table. If an entry for a particular routine already exists, it is replaced.

The following exception errors may be caught by user programs:

Bus error
Address error
Illegal instruction
Zero Divide
CHK instruction
TRAPV instruction
Privilege violation
Line 1010 emulator
Line 1111 emulator

User programs can also catch the following exception errors on systems with a floating point coprocessor (68020 or 68030 with 68881/882; or 68040):

Branch or set on unordered condition
Inexact result
Divide by zero
Underflow
Operand Error
Overflow
NAN signaled

If a user routine is not provided and one of these exceptions occur, the program is aborted. An example initialization table might look like:

```
ExcpTbl   dc.w   T_TRAPV,OvfError-*-4
          dc.w   T_CHK,CHKError-*-4
          dc.w   -1 End of Table
```

When an exception routine is executed, it is passed the following:

d7.w = Exception vector offset
(a0) = Program counter when exception occurred
(same as R$PC(a5))
(a1) = Stack pointer when exception occurred (R$a7(a5))
(a5) = User's register stack image when exception occurred
(a6) = user's primary global data pointer

To return to normal program execution after handling the error, the exception must restore all registers (from the register image at (a5)), and jump to the return program counter. For some kinds of exceptions (especially bus and address errors) this may not be appropriate. It is the user program's responsibility to determine whether and where to continue execution.

It is possible to disable an error exception handler. This is done by calling F$STrap with an initialization table that specifies zero as the offset to the routine(s) that are to be removed. For example, the following table removes user routines for the trapv and chk error exceptions:

```
Table    dc.w  T_TRAPV, 0
         dc.w  T_CHK, 0
         dc.w  -1
```

CAVEATS: Beware of exceptions in exception handling routines. They are usually not re-entrant.

F$SUser Set User ID Number

ASM CALL: **OS9 F$SUser**

INPUT: **d1.l = Desired group/user ID number**

OUTPUT: **None**

ERROR OUTPUT: **cc = Carry bit set**
d1.w = Appropriate error code

FUNCTION: F$SUser alters the current user ID to the specified ID. The following restrictions govern the use of F$SUser:

- User number 0.0 may change their ID to anything without restriction.
- A primary module owned by user 0.0 may change its ID to anything without restriction.
- Any primary module may change its user ID to match the module's owner.

All other attempts to change user ID number return an E$Permit error.

F$SysDbg — Call System Debugger

ASM CALL: **OS9 F$SysDbg**

INPUT: **None**

OUTPUT: **None**

ERROR OUTPUT: **cc = Carry set**
d1.w = Appropriate error code

FUNCTION: F$SysDbg invokes the system level debugger, if one exists, to allow system-state routines, such as device drivers, to be debugged. The system level debugger runs in system state and effectively stops timesharing whenever it is active. It should never be used when there are other users on the system. This call can be made only by a user with a group.user ID of 0.0.

CAVEATS: You must enable the system debugger before installing breakpoints or attempting to trace instructions. If no system debugger is available, the system is reset. The system debugger takes over some of the exception vectors directly, in particular the Trace exception. This makes it impossible to use the user debugger when the system debugger is enabled.

F$Time — Get System Date and Time

ASMCALL: **OS9 F$Time**

INPUT: **d0.w = Format** **0 = Gregorian**
1 = Julian
2 = Gregorian with ticks
3 = Julian with ticks

OUTPUT: **d0.l = Current time**
d1.l = Current date
d2.w = day of week (0 = Sunday to 6 = Saturday)
d3.l = tick rate/current tick (if requested)

ERROR OUTPUT: **cc = Carry bit set**
d1.w = Appropriate error code

FUNCTION: F$Time returns the current system date and time. In the (normal) Gregorian format, time is expressed as 00hhmmss, and date as yyyymmdd. The Julian format expresses time as seconds since midnight, and date as the Julian day number. You can use this to determine the elapsed time of an event. If ticks are requested, the clock tick rate in ticks per second is returned in the most significant word of d3. The least significant word contains the current tick.

The following chart illustrates the values returned in the registers:

	Register	Offset	Gregorian Format	Julian Format
d0.l	**byte**	3	**zero**	**seconds since midnight** (long) 0-86399
		2	**hour** (0-23)	
		1	**minute** (0-59)	
		0	**second** (0-59)	
d1.l	**byte**	2-3	**year** (integer)	**Julian day number** (long)
		1	**month** (1-12)	
		0	**day** (1-31)	

SEE ALSO: F$STime and F$Julian.

CAVEATS: F$Time returns a date and time of zero (with no error) if no previous call to F$STime is made. A tick rate of zero indicates the clock is not running.

F$TLink — Install User Trap Handler Module

ASM CALL: **OS9 F$TLink**

INPUT: **d0.w = User Trap Number (1-15)**
d1.l = Optional memory override
(a0) = Module name pointer
If (a0)=0 or [(a0)]=0, trap handler is unlinked.
Other parameters may be required for specific trap handlers.

OUTPUT: **(a0) = Updated past module name**
(a1) = Trap library execution entry point
(a2) = Trap module pointer
Other values may be returned by specific trap handlers

ERROR OUTPUT: **cc = Carry bit set**
d1.w = Appropriate error code

FUNCTION: You can use user traps as a convenient way to link into a standard set of library routines at execution time. This provides the advantage of keeping user programs small, and automatically updating programs that use the library code if it is changed (without having to re-compile or re-link the program itself). Most Microware utilities use one or more trap libraries.

F$TLink attempts to link, or load, the named module, installing a pointer to it in the user's process descriptor for subsequent use in trap calls. If a trap module already exists for the specified trap code, an error is returned. OS-9 allocates and initializes static storage for the trap handler, if necessary. You can remove traps by passing a null pointer.

A user program calls a trap routine using the following assembly language directive:

```
tcall N,Function
```

This is the equivalent to:

```
trap #N
dc.w Function
```

"N" can be 1 to 15 (specifying which user trap vector to use). The function code is not used by OS-9, except that it is passed to the trap handler, and the program counter is skipped past it.

F$TLink allows the program to delay installation of the handler until a trap is actually used in the program. If a user program executes a user trap call before the corresponding F$TLink call has been made, the system executes the user's default trap exception entry point (specified in the module header) if one exists.

SEE ALSO: Chapter 5 on User Trap Handlers.

CAVEAT: System-state processes should not attempt to use trap handlers.

F$Trans — Translate Memory Address

ASM CALL: **OS9 F$Trans**

INPUT: **d0.l = size of block to translate**
d1.l = mode: 0 - local CPU address to external bus addr
1 - external bus address to local CPU addr
(a0) = address of block

OUTPUT: **d0.l = size of block translated**
(a0) = translated address of block

ERROR OUTPUT: **cc = Carry bit set**
d1.w = Appropriate error code

FUNCTION: On systems with dual-ported memory, a memory location may appear at different addresses depending upon whether it is accessed via the "local" CPU bus or the system's external bus. You can use the F$Trans request to translate an address to or from its external bus address.

F$Trans is used when the external bus address must be passed to hardware devices, such as DMA-type controllers. Using the local CPU bus address is faster and reduces the traffic on the external bus. Generally, you should only use the system's external bus address if it is not possible to use the local CPU bus address.

If the specified source block is non-linear with respect to its destination mapping, F$Trans returns the maximum number of bytes accessible at the translated address. In this case, subsequent calls to F$Trans must be made until the entire block has been successfully translated. This is rare, since OS-9's memory management routines do not allocate non-linear blocks.

SEE ALSO: ***OS-9 Technical Overview***, Chapter 2, sections on Init module memory definitions and Colored Memory.

POSSIBLE ERRORS: E$UnkSvc, E$Param, and E$IBA.

F$UAcct User Accounting

ASM CALL: **OS9 F$UAcct**

INPUT: **d0.w = Function code (F$Fork, F$Chain, F$Exit)**
(a0) = Process descriptor pointer

OUTPUT: **none**

ERROR OUTPUT: **cc = carry bit set**
d1.w = error code if error

FUNCTION: F$UAcct is a user-defined system call which may be installed by an OS9P2 module. It is called in system state at the beginning and end of every process, in other words, whenever F$Fork, F$Chain, or F$Exit is executed.

The kernel's fork and chain routines make an F$UAcct request just before a new process is inserted in the active queue. Since the new process is ready to execute, its user number, priority, primary module, parameters, etc. are known to F$UAcct. This provides a variety of opportunities for a F$UAcct routine. For example:

- A system administrator could keep track of every program run and who ran what program.
- F$UAcct could automatically lower the priority of particular programs.
- F$UAcct could keep a log of everything a specific user does.

NOTE: If F$UAcct returns an error during F$Fork, the new process terminates with the error code in d1.w.

OS-9's process termination routine makes a F$UAcct request just before a process's resources are returned to the system. The process descriptor contains information about how much CPU time was consumed, how many bytes were read or written, how many system calls were made, etc. Once again, F$UAcct could be used to record or react to this information. The system ignores any F$UAcct error returned at the end of a process.

NOTE: The values in all registers except d0 and d1 must be preserved.

SEE ALSO: F$SSvc; ***OS-9 Technical Overview***, Chapter 2 (section on installing system-state routines).

POSSIBLE ERRORS: E$UnkSvc and E$Param.

F$UnLink — Unlink Module by Address

ASM CALL: **OS9 F$UnLink**

INPUT: **(a2) = Address of the module header**

OUTPUT: **None**

ERROR OUTPUT: **cc = Carry bit set**
d1.w = Appropriate error code

FUNCTION: F$UnLink tells OS-9 that the module is no longer needed by the calling process. The module's link count is decremented. When the link count equals zero, the module is removed from the module directory and its memory is de-allocated. When several modules are loaded together as a group, modules are only removed when the link count of all modules in the group have zero link counts.

Device driver modules in use and certain system modules cannot be unlinked.

SEE ALSO: F$UnLoad

CAVEATS: Repetitive UnLink calls to the same module artificially lower its link count, regardless of the number of current users. If the link count becomes zero while the module is being used, it is removed from the module directory and its memory de-allocated. This causes severe problems for whoever is currently using the module, and may crash the system.

F$UnLoad — Unlink Module by Name

ASM CALL: **OS9 F$UnLoad**

INPUT: **d0.w = Module type/language**
(a0) = Module name pointer

OUTPUT: **(a0) = Updated past module name**

ERROR OUTPUT: **cc = Carry bit set**
d1.w = Appropriate error code

FUNCTION: F$UnLoad locates the module in the module directory, decrements its link count, and removes it from the directory if the count reaches zero. Note that this call differs from F$UnLink in that the pointer to the module name is supplied rather than the address of the module header.

SEE ALSO: F$UnLink

CAVEAT: Repetitive UnLoad calls to the same module artificially lower its link count, regardless of how many users are currently using it. If the link count becomes zero while the module is being used, it is removed from the module directory and its memory de-allocated. This causes severe problems for whoever is currently using the module, and may crash the system.

F$Wait — Wait for Child Process to Terminate

ASM CALL: **OS9 F$Wait**

INPUT: **None**

OUTPUT: **d0.w = Terminating child process's ID**
d1.w = Child process's exit status code

ERROR OUTPUT: **cc = Carry bit set**
d1.w = Appropriate error code

FUNCTION: F$Wait causes the calling process to deactivate until a child process terminates by executing a F$Exit system call, or otherwise is terminated. The child's ID number and exit status are returned to the parent. If the child process died due to a signal, the exit status word (register d1) is the signal code.

If the caller has several child processes, the caller is activated when the first one dies, so one Wait system call is required to detect termination of each child.

If a child process died before the Wait call, the caller is reactivated immediately. Wait returns an error only if the caller has no child processes.

SEE ALSO: F$Exit, F$Send, and F$Fork.

CAVEATS: The process descriptors for child processes are not returned to free memory until their parent process does a F$Wait system call or terminates.

If a signal is received by a process waiting for children to terminate, it is activated. In this case, d0.w contains zero, since no child process has terminated.

POSSIBLE ERRORS: E$NoChld

End of Chapter 1

NOTES