OS Support Procedures

Introduction

MLIDs developed using the MSM do not need to make any calls to the operating system. Because the sample code provided shows calls to the operating system, this section is provided as a reference for developers. Novell recommends that developers do not use direct calls to the operating system.

Most of the NetWare OS support routines in this chapter are written in C. The support routine descriptions show the procedure and parameter names in C syntax. Each explanation includes the parameters that must be passed on entry to the routine, the results returned - if any, and an example.

As the examples show, the parameters are placed on the stack in the reverse order of their definition. It is the calling programs responsibility to clean up the stack on return.

As with other NetWare OS routines written in C, the EBX, EBP, ESI, and EDI registers are preserved. Be aware that this is not the case for the assembly language routines.

Note: This appendix is **not** intended to be a comprehensive operating system procedure reference. It simply covers some of the primary routines used by the MSM as an aid to understanding the sample source code.

LONG

AddPollingProcedureRTag (void (*Procedure) (void), struct ResourceTagStructure *RTag);

Parameters

Procedure	Pointer to a polling procedure defined by the driver. The OS calls this procedure at process time.
Rtag	The resource tag acquired by the driver to add the polling procedure.

On Return

EAX	Zero if successful; the polling procedure was added. Otherwise, the procedure failed and the driver should abort initialization.

Description

The driver uses *AddPollingProcedureRTag* to register its polling procedure, when one exists. This routine may only be called at process time, normally during initialization.

After this routine has completed successfully, the operating system continuously calls the procedure specified by the procedure parameter whenever the server has no other work to do. Because this does not guarantee that the procedure will be called within a certain period of time (the operating system may be busy), Novell recommends that the driver also include an interrupt backup procedure to allow the driver to get immediate attention.

There should be only one polling procedure per driver. A single polling procedure should service all physical boards of the same type in the server.

Example

```
push     PollResourceTag            ; polling resource tag
push     OFFSET MyDriverPoll        ; pointer to polling routine
call     AddPollingProcedureRTag
add      esp, 4 * 2                 ; clean up stack
or       eax, eax                   ; check for successful completion
jnz      ErrorAddingPollProcedure   ; handle error if necessary
```

void *

Alloc

(LONG Size,
 struct ResourceTagStructure *RTag);

Parameters

Size	Amount of memory (in bytes) to be allocated
RTag	Resource tag obtained by the driver for memory allocation

On Return

EAX	Zero indicates failure; the routine was unable to allocate memory Non-zero value is a pointer to the allocated memory

Description

Alloc is used to get memory for any driver requirements, such as IOConfigurationStructures or special buffers. This routine can be called at either process or interrupt time. Interrupts may be in any state and will remain unchanged.

The driver passes *Alloc* the amount of memory to be allocated and the routine returns a pointer to the allocated memory. The allocated memory is not initialized.

Example

```
push     AllocSignatureRTag         ; pointer to resource tag
push     SIZE DriverConfigStructure ; amount of memory required
call     Alloc
add      esp, (2 * 4)               ; restore stack
or       eax, eax                   ; check for error allocating memory
jz       ErrorGettingBoardDataSpace ; exit initialization on error
mov      ebx, eax                   ; hold on to pointer to memory
```

void *

AllocateMappedPages

```
( LONG NumberOf4KPages,
  LONG SleepOKFlag,
  LONG Below16MegFlag,
  struct ResourceTagStructure *RTag,
  LONG *SleptFlag );
```

Parameters

NumberOf4KPages	Number of 4K pages to allocate
SleepOKFlag	Set to any non-zero value to allow this call to let other processes execute temporarily if it needs to. If the Below16MegFlag is set, this flag must also be set; otherwise it is optional. The advantage of setting this flag is to allow the OS to rearrange pages if it is unable to find a continuous buffer.
Below16MegFlag	Set if the pages must be physically below the first 16 Megabyte boundary. This is only necessary for intelligent 24-bit adapters that must access memory through a bus mastering device.
RTag	Resource tag obtained by the driver for memory allocation. If the Below16MegFlag is set, the RTag must be obtained using the "CachBelow16MegMemorySignature". Otherwise use the same resource tag as the one used for Alloc.
SleptFlag	Pointer to a dword to be filled in by this procedure that will indicate if the call went to sleep. If this is not needed, set to zero.

On Return

EAX	Zero indicates failure; the routine was unable to allocate memory. A non-zero value points to the allocated memory.

Description

This procedure is used to allocate memory on 4K (page) boundaries and, optionally, to obtain the memory below the 16 megabytes boundary. It is recommended that this procedure be used instead of *AllocBufferBelow16Meg*. It is the responsibility of the driver to return this buffer at shutdown using *DeAllocateMappedPages*.

Call at process time only. Interrupts can be in any state and will not be enabled.

Example

```
push   0                      ;Null slept flag.
push   AllocRTag              ;resource tag
push   0                      ;no 16 meg boundary concerns
push   1                      ;call can sleep if it needs to.
push   (size TableStruct + 4095) SHR 12   ; Round up and convert to pages
call   AllocateMappedPages    ;allocate memory
add    esp, (5*4)             ;clean up stack
or     eax, eax               ;buffer returned?
je     ErrorAllocatingPages   ;jump if not
mov    TablePointer, eax      ;save pointer
```

struct ResourceTagStructure*

AllocateResourceTag

(struct LoadDefinitionStructure *ModuleHandle,
BYTE *ResourceDescriptionString,
LONG ResourceSignature);

Parameters

ModuleHandle	Pointer to the LoadableModuleHandle (passed to the driver's initialization routine on the stack)
ResourceDescriptionString	Pointer to a null-terminated text string describing the resource for which the tag is being allocated. The string can be a maximum of 16 characters including the null. For example: `InterruptDescriptionMsg db 'ACME Driver ISR',0`
ResourceSignature	value identifying a specific resource type. (listed below)

On Return

EAX	Resource tag identifying the specified entry type. A value of zero indicates failure; the operating system did not allocate a resource tag and the driver should abort initialization.

Description

In order for the driver to get resources from the OS, it must first obtain a resource tag. A resource tag is an identifier required by the OS to track system resources.

AllocateResourceTag provides the driver with an operating system resource tag for a specific resource type (refer to the list below). This routine is normally called during initialization and can only be called at process time.

There are unique tags for different types of resources. The driver **must** use the following resource signatures to identify each resource tag type:

```
AESProcessSignature               equ   'PSEA'
AllocSignature                    equ   'TRLA'
CacheBelow16MegMemorySignature    equ   '61BC'
ECBSignature                      equ   'SBCE'
EventSignature                    equ   'TNVE'
InterruptSignature                equ   'PTNI'
IORegistrationSignature           equ   'SROI'
MLIDSignature                     equ   'DILM'
PollingProcedureSignature         equ   'RPLP'
TimerSignature                    equ   'RMIT'
```

Example

```
push     AllocSignature                 ; resource signature (TRLA)
push     OFFSET MemoryRTagMessage       ; resource message
push     [esp + MyHandle + (2 * 4)]     ; module handle
call     AllocateResourceTag
add      esp, (3 * 4)                   ; restore stack
or       eax, eax                       ; allocation successful?
jz       ErrorAllocatingRTag            ; exit init if not
mov      MemoryRTag, eax                ; store pointer to tag
```

void*
AllocBufferBelow16Meg

(LONG RequestedSize
LONG *ActualSize,
struct ResourceTagStructure *RTag);

Parameters

RequestedSize	number of contiguous bytes requested
*ActualSize	pointer to a location where the routine places the actual number of bytes allocated
*RTag	resource tag acquired by driver for memory allocation (with a "CacheBelow16MegMemorySignature")

On Return

EAX	Zero indicates failure; the routine was unable to allocate memory. A non-zero value points to the allocated memory.

Description

AllocBufferBelow16Meg is only used to allocate memory in the case of drivers that support 24-bit host adapters running in machines with more than 16 megabytes of memory. For all other cases drivers must call *Alloc* to allocate required memory.

This call allocates memory so the driver can do I/O operations to or from intermediate buffers below 16 megabytes. The data can then be copied to or from the actual request buffer when it is above the 16 megabyte boundary. The pointer to the buffer allocated is returned in EAX (zero if none allocated). The allocated memory is not initialized.

This call must only be made at process time.

Note: Use these buffers sparingly. The pool of buffers below 16 megabytes is limited to 16. The size of each allocated buffer is equal to the cache buffer size. The default cache buffer size on a server is 4K. For example, if all 16 buffers are allocated using the default cache buffer size, 64K of memory is allocated. The number of buffers in the pool can be set in the STARTUP.NCF file (up to a maximum of 200).

Example: `Set reserved buffers below 16 Meg = 32`

Example

```
push     MemBelow16RTag          ; pointer to resource tag
push     OFFSET ActualSize       ; amount of memory acquired
push     RequestedSize           ; number of bytes required
call     AllocBufferBelow16Meg
add      esp, 3*4                ; adjust stack pointer
or       eax, eax                ; check if successful
jz       ErrorAllocatingMemory   ; jump if error
mov      MyBufferPtr, eax        ; save pointer to allocated memory
```

CancelInterruptTimeCallBack

(an assembly language routine)

On Entry

EDX	contains a pointer to the timer node to be canceled
Interrupts	are disabled

On Return

EDI	is destroyed
ESI	is destroyed
Interrupts	are preserved and were not enabled during the routine

Description

The driver calls *CancelInterruptTimeCallBack* to cancel a call back event previously scheduled using *ScheduleInterruptTimeCallBack*. *CancelInterruptTimeCallBack* removes the specified timer node from the list of events to be called by the timer interrupt handler.

Remember that *ScheduleInterruptTimeCallBack* must be rescheduled after every call back, and that *CancelInterruptTimeCallBack* is only used to cancel a call back if the driver is unloaded before the call back occurs.

Example

```
push     esi                              ; if value must be preserved
push     edi                              ; if value must be preserved
cli
mov      edx, OFFSET MyTimerNode          ; pointer to TimerDataStructure
call     CancelInterruptTimeCallBack
sti
pop      edi                              ; restore original value
pop      esi                              ; restore original value
```

void

CancelNoSleepAESProcessEvent (struct AESProcessStructure *EventNode);

Parameters

EventNode	pointer to the AESProcessEventStructure to be canceled

Description

CancelNoSleepAESProcessEvent removes the specified AESEvent from the operating system's list of events to be called by the AES No-Sleep Process.

This routine may be called at process or interrupt time. Before the driver makes this call, interrupts must be disabled. When the procedure returns, the interrupt state is still disabled and interrupts were not enabled.

Remember that *ScheduleNoSleepAESProcessEvent* must be rescheduled every time it calls the specified process. *CancelNoSleepAESProcessEvent* is called only to cancel a process event if the driver is unloaded before the process executes.

Example

```
cli
push     OFFSET MyAESEventStructure    ; address of AES structure
call     CancelNoSleepAESProcessEvent
add      esp, 4                        ; adjust stack pointer
sti
```

void

CancelSleepAESProcessEvent (struct AESProcessStructure *EventNode);

Parameters

EventNode	pointer to the AESProcessEventStructure to be canceled

Description

CancelSleepAESProcessEvent removes the specified AESEvent from the operating system's list of events to be called by the AES Process.

This routine may be called at process or interrupt time. Before the driver makes this call, interrupts must be disabled. When the procedure returns, the interrupt state is still disabled and interrupts were not enabled.

Remember that *ScheduleSleepAESProcessEvent* must be rescheduled every time it calls the specified process. *CancelSleepAESProcessEvent* is called only to cancel a process event if the driver is unloaded before the process executes.

Example

```
cli
push     OFFSET MyAESEventStructure    ; address of AES structure
call     CancelSleepAESProcessEvent
add      esp, 4                        ; adjust stack pointer
sti
```

LONG

ClearHardwareInterrupt

(LONG HardwareInterruptLevel,
void (*InterruptProcedure)(void));

Parameters

HardwareInterruptLevel	IRQ level of the hardware interrupt
InterruptProcedure	pointer to the interrupt procedure

On Return

EAX	Zero indicates the hardware interrupt was successfully removed. A non-zero value means the routine did not clear the interrupt vector because of invalid parameters or not finding the vector

Description

ClearHardwareInterrupt releases a processor hardware interrupt previously allocated by *SetHardwareInterrupt* for a physical board. This routine must only be called at process time and interrupts must be disabled.

ClearHardwareInterrupt is usually called when the driver is unloading or the initialization procedure fails after an interrupt has been set.

Example

```
cli
push     OFFSET MyInterruptHandler  ; interrupt entry
push     InterruptLevel             ; interrupt number
call     ClearHardwareInterrupt
add      esp, (2 * 4)               ; restore stack
sti
```

void

CPSemaphore (LONG SemaphoreNumber);

Parameters

SemaphoreNumber	pointer to the semaphore

Description

CPSemaphore is used to lock the real mode workspace when making an EISA BIOS call. Interrupts are preserved, but will be disabled during the call.

Do not use this call to handle critical sections local to the driver.

Example

```
push     WorkSpaceSemaphore   ; load semaphore
call     CPSemaphore          ; lock workspace for our use
add      esp, (1 * 4)         ; restore stack
```

void

CRescheduleLast (void);

Description

CRescheduleLast places the task in last place on the list of active tasks to be executed. This routine must only be called from the process level as it will suspend the process and could change the machine state.

CRescheduleLast is normally used in conjunction with AESSleepEvents and should only be used in the initialization or driver remove procedures.

Example

```
call     CRescheduleLast   ; will regain control some undefined time later
```

void
CVSemaphore (LONG SemaphoreNumber);

Parameters

SemaphoreNumber	pointer to the semaphore

Description

CVSemaphore clears a semaphore that was set with *CPSemaphore*. Interrupts are preserved, but will be disabled during the call.

Normally, *CVSemaphore* is used when the driver has finished making an EISA BIOS call so that other processes can be allowed to use the workspace.

Example

```
push     WorkSpaceSemaphore   ;pass semaphore
call     CVSemaphore          ;unlock workspace
add      esp, (1 * 4)         ;restore stack
```

void

DeAllocateMappedPages (void *BufferPointer);

Parameters

*BufferPointer	Pointer to the buffer to free. (must have been allocated with *AllocateMappedPages*)

Description The driver must use this routine to return any memory buffers that were previously allocated on 4K page boundaries using the *AllocateMappedPages* procedure.

Example

```
push    TablePointer          ;pointer to buffer
call    DeAllocateMappedPages ;deallocate memory
add     esp, 1*4              ;clean up stack
```

void
DeRegisterHardwareOptions (struct IOConfigurationStructure *IOConfig);

Parameters

IOConfig	pointer to the physical board's IOConfigurationStructure (starting at the *CDriverLink* field of the configuration table)

Description

DeRegisterHardwareOptions releases the previously reserved hardware options specified in a particular physical board's *IOConfigurationStructure* (starting at the *CDriverLink* field of the configuration table). This procedure must only be called from the process level and must be called with interrupts disabled.

DeRegisterHardwareOptions will usually be made from the driver's remove procedure (or possibly from *Ctl5_MLIDShutdown* if the control procedure is doing a complete shutdown).

Example

```
cli
push     [ebx].CDriverLink           ; pointer to IOConfigurationStructure
call     DeRegisterHardwareOptions
add      esp, 4                      ; restore stack
sti
```

DisableHardwareInterrupt

(an assembly language routine)

On Entry

ECX	contains the interrupt level
Interrupts	should be disabled
Execute	at process or interrupt time

On Return

Interrupts	are unchanged
Note	EAX and EDX are destroyed; all other registers are preserved

Description

This routine masks off the ECX-specified interrupt request line on the programmable interrupt controller, preventing the adapter from interrupting the driver.

This routine is not needed if the adapter runs on an edge-triggered interruptible bus and provides a command to disable its interrupt line.

Note: Novell recommends disabling interrupts at the NIC if possible. Disabling interrupts at the PIC is typically slower.

Example

```
DriverISR     proc

   mov   ecx, InterruptLevel
   call  DisableHardwareInterrupt
   call  DoEndOfInterrupt
    .
    .    (Service the adapter)
    .
   mov   ecx, InterruptLevel
   call  EnableHardwareInterrupt
   call  LSLServiceEvents                 ; Let LSL unqueue returned
   ret

DriverISR     endp
```

DoEndOfInterrupt

(an assembly language routine)

On Entry

ECX	contains the interrupt level
Interrupts	should be disabled
Execute	at process or interrupt time

On Return

Interrupts	are unchanged
Note	EAX is are destroyed; all other registers are preserved

Description

This routine issues the appropriate End of Interrupt (EOI) commands to one or both programmable interrupt controllers (PICs). If the level is assigned to a secondary PIC, an EOI will be issued to the secondary PIC, then to the primary PIC. Use of this routine (instead of hard-coding EOIs in the driver) allows flexibility when a driver runs on several platforms and ensures that this function is executed correctly in the event of future operating system changes.

Example

(see example for *DisableHardwareInterrupt*)

LONG

DoRealModeInterrupt (struct InputParameterStructure *InputParameters, struct OutputParameterStructure *OutputParameters);

Parameters

InputParameters	pointer to a filled in InputParameterStructure defined below
OutputParameters	pointer to a filled in OutputParameterStructure defined below

On Return

EAX	0: if the interrupt vector is called successfully 1: if the call fails because the interrupt vector is no longer available (DOS has been removed)

Description

DoRealModeInterrupt is used to perform real mode interrupts, such as BIOS and DOS interrupts. This routine can only be called at process time, and it may enable interrupts and put the calling process to sleep.

EISA boards will need to use *DoRealModeInterrupt* to perform the INT 15h BIOS call that returns the board configuration. The parameter structures are defined below:

InputParameters

```
InputParameterStructure    struc
   IAXRegister dw ?
   IBXRegister dw ?
   ICXRegister dw ?
   IDXRegister dw ?
   IBPRegister dw ?
   ISIRegister dw ?
   IDIRegister dw ?
   IDSRegister dw ?
   IESRegister dw ?
   IntNumber   db ?
InputParameterStructure    ends
```

OutputParameters

```
OutputParameterStructure   struc
   OAXRegister dw ?
   OBXRegister dw ?
   OCXRegister dw ?
   ODXRegister dw ?
   OBPRegister dw ?
   OSIRegister dw ?
   ODIRegister dw ?
   ODSRegister dw ?
   OESRegister dw ?
   OFlags      dw ?
OutputParameterStructure   ends
```

Example

Note: The input parameter structure has already been initialized.

```
push     OFFSET OutputParameters ; place pointer on stack
push     OFFSET InputParameters  ; place pointer on stack
call     DoRealModeInterrupt
add      esp, 2 * 4              ; clean up stack
cmp      eax, 0                  ; check for error
jne      IntNotValidErrorExit    ; handle error if necessary
```

EnableHardwareInterrupt

(an assembly language routine)

On Entry

ECX	contains the interrupt level
Interrupts	are disabled
Execute	at process or interrupt time

On Return

Interrupts	are unchanged
Note	EAX and EDX are destroyed: all other registers are preserved

Description

This routine enables the adapter's interrupt line on the programmable interrupt controller if *DisableHardwareInterrupt* was previously used.

Example

(see example for *DisableHardwareInterrupt*)

void
Free (void *MemoryBuffer);

Parameters

MemoryBuffer	pointer to the previously allocated memory to be released (Must be memory previously allocated by the Alloc routine)

Description

Free returns the memory previously allocated by the driver for any purpose. This routine may be called at either process or interrupt time. Interrupts can be in any state and that state will be preserved.

Drivers are expected to make this call for all memory that they allocated during initialization, and drivers should always call this routine as an essential part of cleaning up before exiting.

Example

```
push     MyMemoryBlock     ; place pointer to memory on stack
call     Free
add      esp,  1 * 4       ; restore stack
```

void

FreeBufferBelow16Meg (void *MemoryBuffer);

Parameters

MemoryBuffer	pointer to the memory to be returned to NetWare (Must be memory previously allocated by AllocBufferBelow16Meg.)

Description *FreeBufferBelow16Meg* returns the memory previously allocated by the driver for Bus Master or DMA I/O which was required to be below 16 megabytes. Returning memory is an essential part of cleaning up before exiting. This function may be called at process or interrupt time.

Example

```
push     eax                  ; pointer to memory
call     FreeBufferBelow16Meg
lea      esp, [esp +4]        ; adjust stack pointer
```

GetCurrentTime

(an assembly language routine)

On Return

EAX	contains the number of clock ticks (1/18th second or 55.5 milliseconds) since the server was last loaded and began execution.

Description

GetCurrentTime determines the current relative time in order to determine the elapsed time for some driver-related activities (e.g. time out check). The current time value less the value returned at the start of an operation is the elapsed time in 1/18th second clock ticks. It requires more than 7 years for this timer to roll over, allowing it to be used for elapsed time comparisons.

Example

```
mov      edx, [ebp].Command      ; let the board attempt to
mov      al, Board_Transmit      ; transmit packet
out      dx, al

call     GetCurrentTime          ; get current time
mov      [ebp].TxStartTime, eax  ; save for timeout monitoring
```

GetHardwareBusType

(an assembly language routine)

On Return

EAX	0: I/O bus is ISA (Industry Standard Architecture) 1: I/O bus is MCA (Micro-Channel Architecture) 2: I/O bus is EISA (Extended Industry Standard Architecture)

Description

GetHardwareBusType returns a value indicating the processor bus type. This routine may be called at process or interrupt time, and the interrupt state is preserved and will not change.

GetHardwareBusType allows a single driver to be written so that it can be used for boards of different bus types.

Note: These values are different than those used in the *CDriverFlags* field of the configuration table.

Example

```
call     GetHardwareBusType
mov      HardwareBusType, eax ; store returned value
```

GetProcessorSpeedRating

(an assembly language routine)

On Return

EAX	Zero if routine failed to determine the processor speed. Otherwise, EAX contains a value representing the relative processor speed of the machine.

Description

GetProcessorSpeedRating is used to determine the relative processor speed. This routine may be called at process or interrupt time and will not change the interrupt state.

The larger the value returned, the faster the processor can operate. Some drivers may need to use *GetProcessorSpeedRating* to calculate the correct delay for certain timing loops.

Example

```
call     GetProcessorSpeedRating
mov      ProcessorSpeedAdjust, eax  ; save returned processor speed
```

void

GetRealModeWorkSpace

```
( struct SemaphoreStructure *WorkspaceSemaphore,
  LONG   *WorkspaceProtectedModeAddress,
  WORD   *WorkspaceRealModeSegment,
  WORD   *WorkspaceRealModeOffset,
  LONG   *WorkspaceSize );
```

Parameters

WorkspaceSemaphore	pointer to the operating system semaphore structure
WorkspaceProtectedModeAddress	32-bit logical address of the workspace block
WorkspaceRealModeSegment	real mode segment of workspace
WorkspaceRealModeOffset	real mode offset in the workspace segment
WorkspaceSize	size of the workspace

Description

The *GetRealModeWorkSpace* routine is used in conjunction with *DoRealModeInterrupt* to allow the driver access to memory in real mode.

NetWare drivers run in protected mode and do not allow direct access to BIOS based information. The call *DoRealModeInterrupt* allows the driver to access the BIOS.

DoRealModeInterrupt turns on the system interrupts and executes in a critical section; therefore, semaphore routines--*CPSemaphore* and *CVSemaphore* are called in order to keep other processes out of the workspace.

The driver must provide the following variables. On entry, the driver passes this routine pointers to these variables. This routine then fills in the variables with the appropriate values as described above.

```
WorkspaceSemaphore            dd 0
WorkspaceProtectedModeAddress dd 0
WorkspaceRealModeSegment      dw 0
WorkspaceRealModeOffset       dw 0
WorkspaceSize                 dd 0
```

Example

```
;**************************************************************************
; Get  realmode workspace
;**************************************************************************

push     OFFSET WorkSpaceSize                   ; size of workspace
push     OFFSET WorkSpaceRealModeOffset         ; offset to real mode
push     OFFSET WorkSpaceRealModeSegment        ; real mode segment address
push     OFFSET WorkSpaceProtectedModeAddress   ; address in protected mode
push     OFFSET WorkSpaceSemaphore              ; semaphore
call     GetRealModeWorkSpace
add      esp, (5 * 4)                           ; clean up stack

;**************************************************************************
; Lock the workspace
;**************************************************************************

push     WorkSpaceSemaphore                     ; load semaphore
call     CPSemaphore                            ; lock workspace
add      esp, (1 * 4)                           ; clean up stack

;**************************************************************************
; Setup and execute real mode interrupt
;**************************************************************************

movzx    eax, WorkSpaceRealModeSegment          ; get WorkSpace segment
movzx    ebx, WorkSpaceRealModeOffset           ; get offset into segment
mov      cl, SlotToReadConfiguration            ; get slot number
xor      ch, ch                                 ; read first block
mov      esi, OFFSET InputParms                 ; point to input area
mov      [esi].IAXRegister, 0D801h              ; EISA read configuration
mov      [esi].ICXRegister, cx                  ; slot and data block
mov      [esi].ISIRegister, bx                  ; offset of DosWorkarea
mov      [esi].IDSRegister, ax                  ; segment of DosWorkArea
mov      [esi].IIntNumber, 15h                  ; interrupt number
push     OFFSET OutputParms                     ; pointer to output regs
push     OFFSET InputParms                      ; pointer to input regs
call     DoRealModeInterrupt
lea      esp, [esp + 2 * 4]                     ; clear up stack
cmp      eax, 0                                 ; error check
jne      IntNotValidErrorExit                   ; error path
cmp      byte ptr OutputParms.OAXRegister + 1,0 ; BIOS Int 15h return
jne      IntNotValidErrorExit ;successful ?
mov      esi, WorkSpaceProtectedModeAddress     ; load pointer to data
movzx    ecx, BYTE PTR [esi + INTERRUPTOFFSET]  ; get int if any
and      cl, ISOLATEINTMASK                     ; isolate interrupt level
jecxz    NoAddInterrupt                         ; if none skip add
mov      SaveInterrupt, cl                      ; save interrupt for later

;**************************************************************************
; Unlock interrupt
;**************************************************************************

NoAddInterrupt:
push     WorkSpaceSemaphore                     ; pass semaphore
call     CVSemaphore                            ; unlock workspace
add      esp, (1 * 4)                           ; clean up stack
```

GetServerPhysicalOffset

(an assembly language routine)

On Return

EAX	contains a 32-bit physical address

Description

GetServerPhysicalOffset returns the physical address of the operating system's logical address 0. Use this value to convert physical addresses to logical addresses and vice versa. The routine may be called at process or interrupt time. It may be called with the interrupts in any state, and will not change the state.

To find the physical address given a logical offset, add the address this routine returns to the logical address. To find the logical address given a physical address, subtract the value returned from the physical address.

The value that *GetServerPhysicalOffset* returns could be necessary in making address conversions during the initialization of DMA channels and bus mastering devices, and in the validation of specified hardware options.

Example

```
call     GetServerPhysicalOffset
add      esp, 1 * 4
```

LONG

OutputToScreen

(struct ScreenStruct *screenID,
char *controlString,
args...);

Parameters

screenID	ScreenHandle of the console screen which is passed to the driver during initialization
controlString	pointer to a null-terminated ASCII string
args...	procedure can take a variable number of standard Printf control string arguments

On Return

EAX	zero if successful

Description

OutputToScreen is used to display a driver error message on the server console screen. This routine must only be called during initialization at process time. It will not suspend the calling process.

Drivers should not display non-vital messages and should limit the number of lines output to the screen for essential messages as displaying unneeded output will cause important information to scroll off the screen. *controlString* can be embedded with returns, line feeds, bells, tabs and backspaces. However, if strings contain embedded substrings, numbers and control information, they must be limited in length to a maximum of 200 characters as longer strings than this will cause the server to abend. If longer strings are necessary, split the string into several strings and call *OutputToScreen* multiple times.

Note: ScreenID is not valid after returning from the initialization routine, so *OutputToScreen* can only be used during initialization.

Example

```
push     OFFSET MyMessage                      ; push offset to message
push     [esp + InitializationErrorScreen + 4]  ; screen handle
call     OutputToScreen
add      esp, 2 * 4                            ; restore stack
```

LONG

ParseDriverParameters

```
( struct IOConfigurationStructure *IOConfig,
  struct DriverConfigurationStructure *configuration,
  struct AdapterOptionDefinitionStructure *adapterOptions,
  struct LANConfigurationLimitStructure *configLimits,
  BYTE  (*FrameTypeDescription)[ ],
  LONG  needBitMap,
  BYTE  *commandLine,
  struct ScreenStruct *screenID );
```

Parameters

IOConfig	pointer to the Adapter's IOConfigurationStructure (starting at the *CDriverLink* field of the configuration table)
configuration	pointer to the logical board's configuration table
adapterOptions	pointer to the AdapterOptionDefinitionStructure
configLimits	pointer to the LANConfigurationLimitStructure
FrameTypeDescription	pointer to the beginning of an array of pointers to frame descriptors which defines the supported frame type of the packet
needBitMap	bit map telling *ParseDriverParameters* which hardware options the adapter requires
CommandLine	pointer to the command line passed to the driver at load time
ScreenID	pointer to the ScreenHandle which was passed to the driver at initialization

On Return

EAX	Zero: Successful Non-zero: Failed

Description

ParseDriverParameters utilizes the command line parameters, operator input, and the tables provided by the driver to fill in the *IOConfigurationStructure* (starting at the *CDriverLink* field of the configuration table) associated with the configuration table of the logical board. This routine must only be called from the process level as it may suspend the process and could change the machine state. In addition, this routine can only be called at initialization time because *screenID* is only valid at that time.

ParseDriverParameters is used in conjunction with *RegisterHardwareOptions*. Examples of the tables which are provided by the driver are listed below along with the definition of the macro "Message":

FrameDescriptTable

```
FrameDescriptTable
   dd Ethernet8023Descript
   dd EthernetIIDescript
   dd Ethernet8022Descript
   dd EthernetSNAPDescript

   Message  Ethernet8023Descript, ’ETHERNET_ 802.3’
   Message  EthernetIIDescript,   ’ETHERNET_II’
   Message  Ethernet8022Descript, ’ETHERNET_802.2’
   Message  EthernetSNAPDescript, ’ETHERNET_SNAP’
```

Message macro definition

```
Message macro  MessageName, MessageString
                  local StringEnd, StringBegin
   MessageName db StringEnd - StringBegin
   StringBegin db MessageString
   StringEnd   db 0
endm
```

Note: The message macro used above causes the strings in the *FrameDescriptTable* to be length preceded and null terminated.

The *AdapterOptionDefinitionStructure* is a hard coded part of the MLID’s data structure. Using the *NeedsBitMap* as a guide, *ParseIOParameters* collects the necessary information from the command line and from the *AdapterOptionDefinitionStructure*, fills out the appropriate fields in the configuration table and returns successfully.

The driver doesn’t necessarily set the bit in the bitmap field if it uses a parameter; but, if there are multiple possibilities and the driver wants *ParseDriverParameters* (by asking the network supervisor at the console or by parsing the command line) to determine which option to use, it must set the appropriate bit in the *NeedsBitMap*.

Each field in the AdapterOptionDefinitionStructure is a pointer. If the option is not supported, a zero is placed in that field. If an option is supported, a pointer to an option list is placed in that field. The AdapterOptionDefinitionStructure appears as follows:

```
AdapterOptionDefinitionStructure struc

   IOSlot         dd    ?
   IOPort0        dd    ?
   IORange0       dd    ?
   IOPort1        dd    ?
   IORange1       dd    ?
   MemoryDecode0  dd    ?
   MemoryLength0  dd    ?  ; length in bytes
   MemoryDecode1  dd    ?
   MemoryLength1  dd    ?  ; length in bytes
   Interrupt0     dd    ?
   Interrupt1     dd    ?
   DMA0           dd    ?
   DMA1           dd    ?
   Channel        dd    ?

AdapterOptionDefinitionStructure ends
```

Example option list:

```
   IRQOptions        dd 4           ;option count
                     dd 3, 2, 5, 7
   MemoryOptions     dd 2           ;option count
                     dd 0D000h, 0D8000h
   IOPortOptions     dd 4           ;option count
                     dd 300h, 310h, 320h, 330h

   AdapterOptions AdapterOptionDefinitonStructure
            <,IOPortOptions,,,,MemoryOptions,,,,IRQOptions>
```

LAN Configuration Limits

```
MinAddress           db 6  dup   (0)
MaxAddress           db 5  dup   (0FFh), 0FEh

ConfigLimits         label
   MinNodeAddressPtr dd MinAddress
   MaxNodeAddressPtr dd MaxAddress
   MinRetries        dd 0
   MaxCRetries       dd 255
   NumberFrames      dd 4
```

Note: If the driver uses slots, and can scan them at run time to determine which of them hold boards, it should build the appropriate option list without operator intervention.

CanSetNodeAddress or *MustSetNodeAddress* flags must be specified in the *NeedsBitMap* parameter if this option is desired. (These flags were previously in the *NeedFlags* parameter of v3.0.)

NeedsBitMap

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
		0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0														

Bit #	Needs Option	
0	NeedsIOSlotBit	(00000001h)
1	NeedsIOPort0Bit	(00000002h)
2	NeedsIOLength0Bit	(00000004h)
3	NeedsIOPort1Bit	(00000008h)
4	NeedsIOLength1Bit	(00000010h)
5	NeedsMemoryDecode0Bit	(00000020h)
6	NeedsMemoryLength0Bit	(00000040h)
7	NeedsMemoryDecode1Bit	(00000080h)
8	NeedsMemoryLength1Bit	(00000100h)
9	NeedsInterrupt0Bit	(00000200h)
10	NeedsInterrupt1Bit	(00000400h)
11	NeedsDMA0Bit	(00000800h)
12	NeedsDMA1Bit	(00001000h)
13	NeedsChannelBit	(00002000h)
30	CAN_SET_NODE_ADDRESS	(40000000h)
31	MUST_SET_NODE_ADDRESS	(80000000h)

Example

```
push     [esp + InitializationErrorScreen]   ; screen handle
push     [esp + ConfigurationInfo + 4]       ; pointer to command line
push     NeedsIOPort0Bit OR NeedsInterrupt0Bit OR CanSetNodeAddress
push     OFFSET FrameDescriptTable           ; media ID string array
push     OFFSET ConfigLimits                 ; node and Retry limits
push     OFFSET AdapterOptions               ; options to query from user
push     OFFSET DriverConfiguration          ; driver configuration table
push     OFFSET [ebx].CDriverLink            ; IO configuration table
call     ParseDriverParameters
add      esp, 8 * 4                          ; clean up stack
or       eax, eax                            ; successful?
jnz      ErrorParsingDriverOptions           ; exit init if not
```

LONG

QueueSystemAlert

```
( LONG TargetStation,
  LONG TargetNotificationBits,
  LONG ErrorLocus,
  LONG ErrorClass,
  LONG ErrorCode,
  LONG ErrorSeverity,
  BYTE *controlString,
  . . . );
```

Parameters

TargetStation	connection number of the affected station or 0 if no single station is affected (this parameter is usually 0)
TargetNotificationBits	destinations of the notification
ErrorLocus	locus of the error
ErrorClass	class of error
ErrorCode	error codes for the system log
ErrorSeverity	severity of error
controlString	standard Printf control string used in the output routine
...	the routine can take a variable number of standard Printf control string arguments

On Return

EAX	0: Successful 1: Alert Not Available

Description

QueueSystemAlert provides a system notification of driver hardware or software problems during regular operation of the board. This routine may be called at process or interrupt time and will not sleep. When the routine returns, the interrupt states are preserved. If the routine is called with interrupts disabled, interrupts will not have been enabled.

Listed below is a detailed description of each parameter for this call.

TargetStation

This parameter usually holds a zero, which means that no single station is affected.

TargetNotificationBits

```
NOTIFY_CONNECTION_BIT      01h
NOTIFY_EVERYONE_BIT        02h
NOTIFY_ERROR_LOG_BIT       04h
NOTIFY_CONSOLE_BIT         08h
```

ErrorLocus

LOCUS_UNKNOWN	0h
LOCUS_LANBOARDS	4h

ErrorClass

CLASS_UNKNOWN	0h
CLASS_TEMP_SITUATION	2h
CLASS_HARDWARE_ERROR	5h
CLASS_BAD_FORMAT	9h
CLASS_MEDIA_FAILURE	11h
CLASS_CONFIGURATION_ERROR	15h
CLASS_DISK_INFORMATION	18h

ErrorCode

OK	00h
ERR_HARD_FAILURE	0ffh

ErrorSeverity

SEVERITY_INFORMATIONAL	0h
SEVERITY_WARNING	1h
SEVERITY_RECOVERABLE	2h
SEVERITY_CRITICAL	3h
SEVERITY_FATAL	4h
SEVERITY_OPERATION_ABORTED	5h

Example

```
TransmitTimeoutMessage db 'Transmit failure on board #%d', 0


movzx    eax, [ebx].CDriverBoardNumber ; pass the board number
push     eax
push     OFFSET TransmitTimeoutMessage ; pass error string
push     SEVERITY_RECOVERABLE          ; SeverityRecoverable
xor      eax, eax
push     eax                           ; error code
push     CLASS_HARDWARE_ERROR          ; ClassHardwareFailure
push     LOCUS_LANBOARDS               ; LocusLANboards
push     01100b                        ; console & ErrorLog
push     eax                           ; station #, not used
call     QueueSystemAlert
add      esp, 8 * 4                    ; clean up stack
```

ReadEISAConfig

(an assembly language routine)

On Entry

ECX	CH=Block, CL=Slot
Interrupts	may be in any state
Execute	at process time only (typically during initialization)

On Return

EAX	00h = successful 01h = Int 15h vector removed 80h = invalid slot number 81h = invalid function number 82h = nonvolatile memory corrupt 83h = empty slot 86h = invalid BIOS routine called 87h = invalid system configuration
ESI	Pointer to the buffer containing the configuration read
Zero Flag	Set if successful
Interrupts	are preserved but may have been enabled
Note	EDX and EDI are destroyed

Description

This procedure reads the EISA configuration block for the specified slot into a 320-byte buffer. Normally the driver will call this routine with Block = 0. If the information is not found in this block, continue calling this routine and incrementing the Block number until the right block is received (or you run out of blocks).

The configuration block returned should be copied into local memory. Once the driver returns to the operating system or calls a blocking procedure, the block information is no longer valid.

LONG
(*ReadRoutine)

```
( LONG  CustomFileHandle,
  LONG  CustomDataOffset,
  LONG *Destination,
  LONG  CustomDataSize );
```

Parameters

CustomFileHandle	.LAN file's handle, supplied as LoadableModuleFileHandle to the InitializeDriver routine
CustomDataOffset	starting offset in the file, supplied as CustomDataOffset to the InitializeDriver routine
Destination	buffer for file data to be read
CustomDataSize	size of the data to read, supplied as CustomDataSize to the InitializeDriver routine

On Return

EAX	Zero: Successful Non-zero: Failed

Description

ReadRoutine allows drivers to read custom data or firmware that may be required by specific LAN drivers into system memory during initialization. This routine can only be accessed during initialization. Before this routine is called, memory for the file to be read needs to be allocated. This routine may go to sleep and interrupts may be enabled on return.

The entry point of the *ReadRoutine* is not exported by the operating system. The only place it is valid is in the initialization routine. In fact, the entry point is passed as a local parameter (&ReadRoutine) and must be called indirectly.

The NLM linker actually appends the custom data file to the driver in the .LAN file. NetWare only loads the driver's code data at load time, leaving the file open for the driver to handle custom data however it wants.

To define the custom file, use the CUSTOM key word in the driver definition file followed by the file's name. Netware passes the custom file's handle, starting address, and size to the initialization routine. NetWare also passes the address of the *ReadRoutine*. The driver's initialization routine can then read the file into memory by calling the *ReadRoutine*.

The driver must supply the destination in memory according to the needs of the board.

Example

```
mov      eax, dword ptr [esp + CustomDataSize]  ; get size of firmware
push     MemoryRTag                             ; push tag
push     eax                                    ; push size
call     Alloc                                  ; allocate memory to
add      esp, 2 * 4                             ; clean up stack
or       eax, eax                               ; did we get it?
jz       ErrorGettingExtraMemory                ; error exit if not

mov      FirmWareBufferPtr, eax                    ; save firmware buffer
mov      esi, eax                               ; allocated memory
mov      eax, [esp + LoadableModuleFileHandle]  ; file handle firmware
mov      ebx, [esp + &ReadRoutine]              ; read routine address
mov      edx, [esp + CustomDataOffset]          ; start address in file
mov      ecx, [esp + CustomDataSize]            ; get size of firmware
push     ecx                                    ; amount to read
push     esi                                    ; where to read to
push     edx                                    ; offset in file
push     eax                                    ; file handle
call     ebx                                    ; call read routine
cli                                                   ;stop interrupts
add      esp, 4 * 4                             ; adjust the stack
or       eax, eax                               ; check for read
jnz      ReadError                              ; errors
```

Note: The "custom" key word must be used in the definition file to specify the file name for the firmware.

LONG

RegisterForEventNotification

```
( struct ResourceTagStructure *resourceTag,
  LONG eventType,
  LONG priority,
  void (*warnProcedure) (void (*OutputRoutine)(BYTE *controlString,...), LONG parameter),
  void (*reportProcedure)(LONG parameter) );
```

Parameters

ResourceTag	resource tag which is acquired by the driver for event notification
eventType	type of event for which notification is desired
priority	order in which registered call back routines will be called
warnProcedure	pointer to a call back routine which will be called when EventCheck is called
OutputRoutine	used to warn the user against a particular event
controlString	standard Printf control string used in the output routine
...	additional parameters may be passed to the output routine in order to match the control string requirements
parameter	32 bit value which is defined according to the event type
reportProcedure	pointer to a call back routine that is called when EventReport is called

On Return

EAX	Zero: Fail Non-zero: Successful; EAX contains an EventID that should be used when *UnRegisterEventNotification* is called.

Description

RegisterForEventNotification is called at initialization in order to register an event call back routine. For example, the driver calls this routine so that it can be notified if the server is going to exit to DOS. This gives the driver a chance to cancel any AES or timer events and allows bus master devices to return pre-allocated resources and shutdown the adapter.

This procedure will add routines to the event list when an event is reported. These routines will be called according to priority. The warning routine will be called when an EventCheck is called by the operating system, and the report routine will be called when an EventReport is called by the operating system. The parameter passed in when the event is reported will be passed to the routine when it is called. This routine will return an EventID that should be used when *UnRegisterEventNotification* is called.

When the type of event (defined by eventType) occurs, the operating system calls the call back routine. The type of events which may be defined in eventType are listed below:

`EVENT_DOWN_SERVER              4h`
The warn routine and the report routine will be called before the server is shut down. The parameter value is not used.

`EVENT_CHANGE_TO_REAL_MODE      5h`
The report routine will be called before the server changes to real mode and must not go to sleep. The parameter value is not used.

`EVENT_RETURN_FROM_REAL_MODE    6h`
The report routine will be called after the server returns from DOS and must not go to sleep. The parameter value is not used.

`EVENT_EXIT_TO_DOS              7h`
The report routine will be called before the server exits to DOS. The parameter value is not used.

The order in which the call back routines will be called is determined by the priority parameter. Higher priority routines (indicated with a lower number in the priority parameter) are notified first. The available priorities are listed below:

EVENT_PRIORITY_OS	00h
EVENT_PRIORITY_APPLICATION	20h
EVENT_PRIORITY_DEVICE	40h

The call back routines will be passed a parameter, as well as a report routine to be used to warn the user against the occurrence of a particular event. Nulls may be passed to the routine. The parameter reportProcedure will be passed a parameter containing additional event specific information when it is needed.

Example

```
push     OFFSET ExitOSEvent            ;Address of exit routine
push     0
push     EVENT_PRIORITY_OS             ;Set priority level
push     EVENT_EXIT_TO_DOS             ;Set what event
push     EventResourceTag              ;Resource event tag
call     RegisterForEventNotification
add      esp, 4 * 5                    ;Clear up stack
or       eax, eax                      ;Did OS patch in call?
jz       EventPatchError               ;Error did not add procedure
mov      EventID,eax
```

The driver calls *RegisterForEventNotification* so it can be notified if the server exits to DOS. This will give the driver a chance to service the physical board before the OS exits to DOS. This is especially important for physical boards that use DMA or are bus master devices which need to be shutdown to prevent them from writing to memory after DOS gets control.

LONG
RegisterHardwareOptions

(struct IOConfigurationStructure *IOConfig,
struct DriverConfigurationStructure *configuration);

Parameters

IOConfig	pointer to the CDriverLink field in the logical board's configuration table
configuration	pointer to the logical board's configuration table

On Return

EAX	=0: Success; a new adapter was registered. =1: Success; a new frame type was registered. =2: Success; a new channel (multichannel adapters) was registered. >2: The routine failed to register the hardware because of either a conflict or a bad parameter.

Description

RegisterHardwareOptions reserves hardware options for a particular physical board. This routine must only be called from the process level and will not sleep. It can be called from any interrupt state and it will not change that state.

RegisterHardwareOptions should be passed a pointer to an *IOConfigurationStructure* (starting at the *CDriverLink* field of the configuration table) with the specified hardware options to reserve. If any of the hardware options are already in use, the routine returns an error code.

Example

```
  push   OFFSET [ebx].CDriverSignature
  push   OFFSET [ebx].CDriverLink
  call   RegisterHardwareOptions    ;Register hardware
  add    esp, 2 * 4                 ;Now restore stack

  cmp    eax, 2
  ja     ErrorRegisteringHardware
  je     NewChannel
  cmp    eax, 1
  je     NewFrame
;;jmp    NewAdapter
```

void
RemovePollingProcedure (void (*Procedure)(void));

Parameters

Procedure	pointer to a previously added polling procedure

Description

RemovePollingProcedure is used to remove a driver's poll routine from the server's list of polling procedures. This routine may only be called at process time and will not sleep. Interrupts can be in any state and that state will not be changed.

RemovePollingProcedure should be called when a polled driver unloads.

Example

```
push     OFFSET NewDriverPoll    ;Remove us from poll
call     RemovePollingProcedure  ;List
add      esp, 4
```

ScheduleInterruptTimeCallBack

(an assembly language routine)

On Entry

EDX	points to a timer node data structure
Interrupts	are disabled
Call	at process or interrupt time

On Return

Interrupts	interrupts are preserved and are not enabled
Note	EBX and EBP are preserved; assume all other registers are destroyed.

Description

ScheduleInterruptTimeCallBack is used to add an event to the list of events that will be called by the timer interrupt handler. The specified procedure will only be called once, and the driver must call *ScheduleInterruptTimeCallBack* each time it wants a call back. This process does not relinquish control of the CPU.

The TimerNodeDataStructure is shown below:

```
TimerNodeDataStructure  struc
   TLINK                     dd
   TCallBackProcedure        dd ;Set by caller
   TCallBackEBXParameter     dd ;Set by caller
   TCallBackWaitTime         dd ;Set by caller
   TResourceTag              dd ;Set by caller
   TReserved1                dd
   TReserved2                dd
TimerNodeDataStructure  ends
```

The appropriate fields of this structure should be filled out as follows:

TCallBackProcedure
A pointer to the procedure to be called by the timer interrupt handler. When the procedure is called, interrupts are disabled.

TCallBackEBXParameter
The value EBX should contain when the call back procedure is invoked.

TCallBackWaitTime
The amount of time, in ticks, before the call back procedure is invoked.

TResourceTag
The resource tag the driver allocated in order to use this call

The four fields described above are not changed by the operating system. If the driver reschedules another call back, it does not need to reset these fields.

Example

```
cli
mov      edx, OFFSET MyTimerNode                      ;TimerNodeDataStructu
                                                      re
mov      [edx].TCallBackEBXParameter, ebp             ;Save AdapterPoint
mov      ebx, OFFSET MyTimerInterruptCallBackRoutine
mov      [edx].TCallBackProcedure, ebx
mov      ebx, TimerResourceTag
mov      [edx].TResourceTag, ebx
mov      [edx].TCallBackWaitTime, 5                   ;Wake up in 5 ticks
call     ScheduleInterruptTimeCallBack
```

Note: TResourceTag points to the resource tag acquired by the driver for InterruptTimeCallBacks

void
ScheduleNoSleepAESProcessEvent (struct AESProcessStructure *EventNode);

Parameters

EventNode	pointer to an AESProcessStructure

Description

ScheduleNoSleepAESProcessEvent sets up a background AESNoSleep (AsynchronousEventScheduler) process that will be executed at a desired interval. This procedure can be called at process time or interrupt time. The scheduled procedure will be called at process time and will not relinquish control. When the procedure returns, the interrupt state is preserved and will not have been changed.

ScheduleNoSleepAESProcessEvent will only execute the scheduled procedure once. The driver must call *ScheduleNoSleepAESProcessEvent* every time it wants to execute the procedure.

The driver must have allocated the structure prior to the first call and must have provided the execution level and execution address.

The AESProcessStructure is defined below:

```
AESLink              dd 0
AESWakeUpDelayAmount dd 0
AESWakeUpTime        dd 0
AESProcessToCall     dd 0
AESRTag              dd 0
AESOldLink           dd 0
```

The fields that need to be filled out by the caller in the AESProcessStructure are not changed by the operating system and do not need to be reset if the driver schedules the process again.

Example

```
push     eax                                 ;Points to an AES structure
call     ScheduleNoSleepAESProcessEvent
add      esp, 4                              ;Adjust the stack pointer
```

void
ScheduleSleepAESProcessEvent (struct AESProcessStructure *EventNode);

Parameters

EventNode	pointer to an AESProcessStructure.

Description

ScheduleSleepAESProcessEvent sets up a background AES (Sleep Asynchronous Event Scheduler) thread that will be executed at a desired interval and can be blocked or can make blocking calls while executing. This procedure can be called at process time or interrupt time. The scheduled process will be called at process time and may relinquish control. When the procedure returns, the interrupt state is preserved and will not have been changed.

The scheduled procedure (or thread) will only be executed once. The driver must call *ScheduleSleepAESProcessEvent* each time it wants to execute the procedure (or thread).

The driver must have allocated the structure prior to the first call, and must have provided the execution interval and execution address. A single call to this routine will cause a single entry to the defined routine.

The AESProcessStructure is defined in *ScheduleNoSleepAESProcessEvent*.

Example

```
push     eax                              ;Points to an AES structure
call     ScheduleSleepAESProcessEvent
add      esp, 4                           ;Adjust the stack pointer
```

LONG SetHardwareInterrupt

```
( LONG hardwareInterruptLevel,
  void (*InterruptProcedure) (void),
  struct ResourceTagStructure *RTag,
  LONG endOfChainFlag,
  LONG shareFlag,
  LONG *EOIFlag );
```

Parameters

HardwareInterruptLevel	hardware interrupt level
InterruptProcedure	pointer to the interrupt procedure that will be assigned to the specified interrupt vector
RTag	pointer to ResourceTag acquired by the driver for interrupts
endOfChainFlag	flag which indicates whether chained interrupts are to be placed on the front or the back of the queue by the ISR
shareFlag	flag which indicates whether interrupts may be shared by the device and the driver with other boards and drivers
EOIFlag	pointer to a double-word flag indicating whether a second EOI will be required for this interrupt

On Return

EAX	0: Successful 1: Invalid parameter 2: Invalid sharing mode 3: Out of memory

Description

SetHardwareInterrupt allocates the specified interrupt and provides an ISR entry point. This procedure must only be called from the process level, and it will not suspend the calling process. The interrupts must be disabled, and it will not enable interrupts.

The interrupt procedure will be called with all the registers preserved, ES and DS initialized, and the direction flag cleared. Because interrupt procedures are called as a near procedure, they should return using a RET.

This routine uses three flags:

endOfChainFlag
If this flag is equal to 0, the ISR is to be placed on the front of the queue (non-shared interrupts should use 0). If this flag is equal to 1, and the shareFlag is also equal to 1, the ISR should be placed at the end of the queue.

shareFlag
If this flag is equal to 0 the interrupt is non-sharable. If the flag is equal to 1, the interrupt can be shared.

EOIFlag
If this flag returns with a 0, only one EOI will be required for this interrupt. This flag will be initialized by SetHardwareInterrupt. If this flag is not 0, the interrupt is chained, and the second PIC will also need an EOI. Always EOI the slave (or secondary) PIC first, and then EOI the master (or primary) PIC second.

Example

```
push     OFFSET EOIFlag
push     0                              ;Non sharable interrupt
push     0                              ;End of Chain Flag
push     InterruptResourceTag           ;Pointer to RTag
push     OFFSET MyInterruptHandler
push     MyInterruptLevel               ;Interrupt entry

call     SetHardwareInterrupt           ;Get interrupt back
add      esp, (6 * 4)                   ;Interrupt number
or       eax, eax                       ;Error getting interrupt
jnz      MLIDResetExit                  ;Exit if so
:
:

MyInterruptHandler   proc  near
:
:
ret
MyInterruptHandler   endp
```

LONG UnRegisterEventNotification (LONG eventID);

Parameters

eventID	value which is returned from RegisterForEventNotification

On Return

EAX	0: Successful 1: Fail

Description

UnRegisterEventNotification should be called to unhook the driver from event notification. This routine should be called when the driver is being unloaded.

Note: Do NOT call this routine from within the routine that was called by *RegisterforEventNotification*.

Example

```
push     EventID                        ;Unhook from OS exit
call     UnRegisterEventNotification    ;Call OS to unhook
add      esp, 4                         ;Clear stack
```