5

NetWare SFT III Support Routines

Introduction

This chapter outlines the terminology and conventions used in the support routine listing and details those support routines available to MSL drivers.

Most of the NetWare OS support routines in this chapter are written in C. The 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 routine's 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 by the support routine. Be aware that this is not the case for the register-based routines.

Conventions

In the support routine descriptions, important terms are used which must be understood to design a driver to work properly with NetWare. These terms are defined below:

Interrupts Disabled
: Indicates that interrupts must be disabled when calling the procedure. This means that no processor interrupts (excepting Non-Maskable Interrupts) can occur. This state is often required to maintain system and driver integrity. If not specified, interrupts may be either enabled or disabled when calling the procedure.

Interrupts Enabled
: Indicates that interrupts must be enabled when calling the procedure. This means that processor interrupts can occur. This state is sometimes required to ensure system and driver interruptibility. If not specified, interrupts may be either enabled or disabled when calling the procedure. Also, unless specifically indicated otherwise, the interrupt enable/disable state is maintained during the call and returned in the same state to the caller.

Blocking
: Indicates the routine may cause the current thread of execution (NetWare process) to be suspended (blocked) until a requested function is completed (or calls other blocking system routines). At no time can a driver's ISR make a call to a blocking routine.

Non-Blocking
: Indicates the routine will run to completion without causing the current thread or process to be suspended.

Process Level

Indicates the level of execution of NetWare 386 processes or scheduled tasks. NLMs normally execute at process level. Also, the loader and command processor execute at process level.

Interrupt Level

Indicates an execution level caused by a processor interrupt. ISRs executes under the identity of the process whose execution was interrupted). Because the current process is unknown, ISRs cannot make calls to any blocking level routines.

By the above definitions, all routines shown as blocking may only be called from blocking process level. Also, all routines shown as non-blocking may be called from either blocking or non-blocking process levels. (see Chapter 2 for more information on execution levels)

Support Routines

Register-based Routines

- CancelInterruptTimeCallBack
- DisableHardwareInterrupt
- DoEndOfInterrupt
- EDXCallBackProcedure
- EnableHardwareInterrupt
- GetNextPacketPointer
- ReadEISAConfig
- ReceiveServerCommPointer
- ScheduleInterruptTimeCallBack
- SendServerCommCompletedPointer

Stack-based Routines

- AddPollingProcedureRTag
- Alloc
- AllocateMappedPages
- AllocateResourceTag
- AllocBufferBelow16Meg
- AllocSemiPermMemory
- CancelNoSleepAESProcessEvent
- CancelSleepAESProcessEvent
- CCheckHardwareInterrupt
- CDisableHardwareInterrupt
- CDoEndOfInterrupt
- CEnableHardwareInterrupt
- ClearHardwareInterrupt
- CPSemaphore
- CRescheduleLast
- CVSemaphore
- DeAllocateMappedPages
- DelayMyself

- DeRegisterHardwareOptions
- DeRegisterServerCommDriver
- DoRealModeInterrupt
- Free
- FreeBufferBelow16Meg
- FreeSemiPermMemory
- GetCurrentTime
- GetHardwareBusType
- GetProcessorSpeedRating
- GetRealModeWorkSpace
- GetServerPhysicalOffset
- GetSharedMemoryLinearAddress
- OutputToScreen
- ParseDriverParameters
- QueueSystemAlert
- ReadRoutine
- RegisterForEventNotification
- RegisterHardwareOptions
- RegisterServerCommDriver
- RemovePollingProcedure
- ReturnSharedMemoryLinearAddress
- ScheduleNoSleepAESProcessEvent
- ScheduleSleepAESProcessEvent
- ServerCommDriverError
- SetHardwareInterrupt
- UnRegisterEventNotification

Be aware of the following when coding your MSL driver.

These routines will be phased out after SFT III OS version 3.11

- AllocSemiPermMemory
- FreeSemiPermMemory
- MapAbsoluteAddressToCodeOffset
- MapAbsoluteAddressToDataOffset
- MapCodeOffsetToAbsoluteAddress
- MapDataOffsetToAbsoluteAddress

These routines are available for SFT III OS versions later than 3.11

- Alloc
- AllocateMappedPages
- DeAllocateMappedPages
- DisableHardwareInterrupt
- DoEndOfInterrupt
- EnableHardwareInterrupt
- Free
- GetServerPhysicalOffset
- GetSharedMemoryLinearAddress
- ReadEISAConfig
- ReturnSharedMemoryLinearAddress

AddPollingProcedureRTag

[Blocking]

Syntax

```
long   AddPollingProcedureRTag  (
                void   ( *DriverPollProcedure ) ( void ) ,
                struct ResourceTagStructure   *ResourceTag ) ;
```

Parameters

DriverPollProcedure
Pointer to a polling procedure defined by the driver. The OS calls this procedure at process time.

ResourceTag
Resource tag with a *PollingProcedureSignature* obtained by the driver to register its polling procedure.
(see the *AllocateResourceTag* procedure)

Return Value

EAX is zero if successful (the polling procedure was added), otherwise the procedure failed and the driver should abort initialization.

Requirements

This routine may only be called at process time, normally during initialization.

Description

The driver uses *AddPollingProcedureRTag* to register its polling procedure, when one exists.

After this routine has completed successfully, the operating system continuously calls the procedure specified by *DriverPollProcedure* 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), the driver also should include a backup interrupt 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 MyDriverPollProc       ; pointer to polling routine
call  AddPollingProcedureRTag
add   esp, 2*4                      ; clean up stack
or    eax, eax                      ; check for successful completion
jnz   ErrorAddingPollProcedure      ; handle error if necessary
```

Alloc

[Non-Blocking]

Syntax

```
void  *Alloc  ( long   MemorySize ,
                struct ResourceTagStructure   *ResourceTag ) ;
```

Parameters

MemorySize
Amount of memory (in bytes) to be allocated.

ResourceTag
Resource tag with an *AllocSignature* obtained by the driver for memory allocation. (see the *AllocateResourceTag* procedure)

Return Value

EAX points to the allocated memory. A value of zero indicates the routine failed to allocate the requested memory.

Requirements

This routine can be called at either process or interrupt time. Interrupts may be in any state and will remain unchanged.

Description

Alloc is used to dynamically allocate memory required by the driver. 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.

Memory allocated with this routine should be returned before the driver is removed using the *Free* routine.

Example

```
push  AllocResourceTag              ; pointer to resource tag
push  MyBufferSize                  ; amount of memory required

call  Alloc                         ; allocate the memory

add   esp, 2*4                      ; restore stack
or    eax, eax                      ; check for error allocating memory
jz    ErrorAllocatingMemory         ; jump if error
mov   MyBufferPtr, eax              ; save pointer to allocated memory
```

See Also

Free
AllocBufferBelow16Meg, FreeBufferBelow16Meg
AllocateMappedPages, DeAllocateMappedPages
AllocateResourceTag

AllocateMappedPages

Syntax

```
void  *AllocateMappedPages  (
                    long  NumberOf4KPages,
                    long  SleepOKFlag,
                    long  Below16MegFlag,
                    struct ResourceTagStructure  *ResourceTag,
                    long *SleptFlag );
```

Parameters

NumberOf4KPages
Number of 4K pages to allocate.

SleepOKFlag
Set to any non-zero value to allow this call to sleep (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.

ResourceTag
Resource tag with an *AllocSignature* obtained by the driver for memory allocation. (The same resource tag used for the *Alloc* routine can also be used for this routine.)

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.

Return Value

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

Requirements

This routine must only be called at process time. Interrupts may be in any state and will remain unchanged.

Description

AllocateMappedPages is used to allocate memory on 4K (page) boundaries and, optionally, to obtain the memory below the 16 megabyte boundary. We recommend that this procedure be used instead of *AllocBufferBelow16Meg*.

Memory allocated with this routine should be returned before the driver is removed using *DeAllocateMappedPages*.

Example

```
push   0                              ;null slept flag
push   AllocResourceTag               ;resource tag
push   0                              ;no 16 meg boundary concerns
push   1                              ;call can sleep if it needs to
push   (MyBufferSize + 4095) SHR 12 ;convert to 4K pages

call   AllocateMappedPages            ;allocate memory

add    esp, 5*4                       ;clean up stack
or     eax, eax                       ;buffer returned?
jz     ErrorAllocatingPages           ;jump if not
mov    MyBufferPtr, eax               ;save pointer
```

See Also

DeAllocateMappedPages
AllocateResourceTag
Alloc, Free
AllocBufferBelow16Meg, FreeBufferBelow16Meg

AllocateResourceTag

[Blocking]

Syntax

```
struct ResourceTagStructure   *AllocateResourceTag  (
            struct LoadDefinitionStructure   *ModuleHandle ,
            byte  *ResourceDescriptionString ,
            long   ResourceSignature ) ;
```

Parameters

ModuleHandle
The value of the *ModuleHandle* that was passed on the stack to the driver when its initialization routine was called.

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:

```
MSLRTagMessage   db  'ACME MSL Driver',0
```

ResourceSignature
Value identifying a specific resource type. (listed below)

Return Value

EAX points to a resource tag structure 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.

Requirements

This routine must only be called from a blocking process level (normally during 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). 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'
MSLSignature                     equ   'DLSM'
PollingProcedureSignature        equ   'RPLP'
SemiPermMemorySignature          equ   'EMPS'
TimerSignature                   equ   'RMIT'
```

Example

```
DriverInitialize  proc

   CPush
   mov   ebp, esp
   pushfd
   cli
   .
   .
   .
   push  MSLSignature                  ; resource signature ('DLSM')
   push  OFFSET MSLRTagMessage         ; resource message
   push  [ebp + Parm0]                 ; module handle

   call  AllocateResourceTag

   add   esp, 3*4                      ; restore stack
   or    eax, eax                      ; allocation successful?
   jz    ErrorAllocatingRTag           ; jump if error getting resource tag
   mov   MSLResourceTag, eax           ; store pointer to tag
```

See Also *DriverInitialize*

AllocBufferBelow16Meg

[Non-Blocking]

Syntax

```
void   *AllocBufferBelow16Meg  (
               long   RequestedSize ,
               long  *ActualSizePtr ,
               struct ResourceTagStructure  *ResourceTag ) ;
```

Parameters

RequestedSize
Amount of contiguous memory in bytes requested.

ActualSizePtr
Pointer to a location where this routine places the actual number of bytes allocated.

ResourceTag
Resource tag with a *CacheBelow16MegMemorySignature* obtained by the driver for memory allocation. (see *AllocateResourceTag*)

Return Value

EAX points to the allocated memory. A value of zero indicates the routine failed to allocate the memory.

Requirements

This routine must only be called at process time. Interrupts may be in any state and will remain unchanged.

Description

AllocBufferBelow16Meg is used to allocate memory below the 16 megabyte boundary. The allocated memory is not initialized.

This routine allows drivers to obtain an intermediate transfer buffer for 24-bit bus master/DMA adapters running in machines with more than 16 megabytes of memory. The buffer is then used to handle all I/O data transfers whenever the actual data source or destination is above 16 megabytes. For all other cases, drivers should call *Alloc* to obtain the required memory.

Memory allocated with this routine should be returned before the driver is removed using *FreeBufferBelow16Meg*.

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.

Example

```
push  Below16MegResourceTag          ; pointer to resource tag
push  OFFSET ActualSize              ; amount of memory acquired
push  MyBufferSize                   ; number of bytes required

call  AllocBufferBelow16Meg

add   esp, 3*4                       ; restore stack pointer
or    eax, eax                       ; check if successful
jz    ErrorAllocatingBuffer          ; jump if error allocating memory
mov   MyBufferPtr, eax               ; save pointer to allocated memory
```

See Also *FreeBufferBelow16Meg*
AllocateMappedPages, DeAllocateMappedPages
Alloc, Free
AllocateResourceTag

AllocSemiPermMemory

[Non-blocking]

Syntax

```
void   * AllocSemiPermMemory  (
                long NumberOfBytes ,
                struct ResourceTagStructure  *ResourceTag ) ;
```

Parameters

NumberOfBytes
The amount of memory (in bytes) to be allocated.

ResourceTag
Resource tag with a *SemiPermMemorySignature* obtained by the driver for memory allocation. (see *AllocateResourceTag*)

Return Value

EAX is a pointer to the allocated memory or 0 if unsuccessful.

Requirements

This routine must only be called at process time. It is typically used by drivers for initialization and may not be called from the interrupt level.

Description

AllocSemiPermMemory is used to allocate a block of returnable memory required by the driver. The driver passes *AllocSemiPermMemory* the amount of memory to be allocated and the routine returns a pointer to the allocated memory. The allocated memory is not initialized.

Memory allocated with this routine should be returned before the driver is removed using *FreeSemiPermMemory*.

Example

```
push  SPMemResourceTag          ;resource tag
push  MyBufferSize              ;amount of memory required

call  AllocSemiPermMemory       ;returns pointer to memory in eax

add   esp, 2 * 4                ;clean up stack
or    eax, eax                  ;check for error
jz    Error                     ;jump on error
mov   MyBufferPtr, eax          ;save pointer to memory
```

See Also

FreeSemiPermMemory
Alloc, Free
AllocateResourceTag

CancelInterruptTimeCallBack

[Non-Blocking, Register-Based Routine]

On Entry

EDX must point to the *TimerDataStructure* corresponding to the interrupt time callback event to be cancelled.

On Return

Assume all registers are destroyed.

Requirements

This routine can be called at either process or interrupt time. Interrupts must be disabled on entry and will remain disabled.

Description

The driver calls *CancelInterruptTimeCallBack* to cancel a callback event previously scheduled using *ScheduleInterruptTimeCallBack*. This routine removes the specified timer node from the list of events to be called by the timer tick interrupt handler. If this routine is called but an event was not scheduled, the OS just returns.

Remember that interrupt level callbacks must be rescheduled using *ScheduleInterruptTimeCallBack* after each callback occurs, and that this routine is normally only used to cancel a scheduled callback if it has not yet occurred.

Example

```
pushfd                                  ; save interrupt state
cli                                     ; disable interrupts
mov   edx, OFFSET MyTimerNode           ; pointer to TimerDataStructure
call  CancelInterruptTimeCallBack
popfd                                   ; restore interrupt state
```

See Also

ScheduleInterruptTimeCallBack
ScheduleNoSleepAESProcessEvent, CancelNoSleepAESProcessEvent
ScheduleSleepAESProcessEvent, CancelSleepAESProcessEvent

CancelNoSleepAESProcessEvent

[Non-Blocking]

Syntax

```
void   CancelNoSleepAESProcessEvent  (
            struct AESEventStructure   *EventNode ) ;
```

Parameters

EventNode
Pointer to the *AESEventStructure* corresponding to the non-blocking process level callback event to be cancelled.

Return Value

None

Requirements

This routine may be called at either process or interrupt time. Interrupts must be disabled on entry and will remain disabled.

Description

The driver calls *CancelNoSleepAESProcessEvent* to cancel a callback event previously schedule using *ScheduleNoSleepAESProcessEvent*. This routine removes the specified non-blocking AES event node from the list of events to be called by the AES process handler.

Remember that non-blocking process level callbacks must be rescheduled using *ScheduleNoSleepAESProcessEvent* after each callback occurs, and that this routine is normally only used to cancel a scheduled callback if it has not yet occurred. If this routine is called but an event was not scheduled, the OS just returns.

Example

```
pushfd                                ; save interrupt state
cli                                   ; disable interrupts
push  OFFSET MyAESEventStructure      ; address of AES structure
call  CancelNoSleepAESProcessEvent
add   esp, 1*4                        ; adjust stack pointer
popfd                                 ; restore interrupt state
```

See Also

ScheduleNoSleepAESProcessEvent
ScheduleSleepAESProcessEvent, CancelSleepAESProcessEvent
ScheduleInterruptTimeCallBack, CancelInterruptTimeCallBack

CancelSleepAESProcessEvent

[Non-Blocking]

Syntax

```
void   CancelSleepAESProcessEvent  (
                struct AESEventStructure   *EventNode ) ;
```

Parameters

EventNode
Pointer to the *AESEventStructure* corresponding to the blocking process level callback event to be cancelled.

Return Value

None

Requirements

This routine may be called at either process or interrupt time. Interrupts must be disabled on entry and will remain disabled.

Description

The driver calls *CancelSleepAESProcessEvent* to cancel a callback event previously schedule using *ScheduleSleepAESProcessEvent*. This routine removes the specified blocking AES event node from the list of events to be called by the AES process handler.

Remember that blocking process level callbacks must be rescheduled using *ScheduleSleepAESProcessEvent* after each callback occurs, and that this routine is normally only used to cancel a scheduled callback if it has not yet occurred. If this routine is called but an event was not scheduled, the OS just returns.

Example

```
pushfd                                  ; save interrupt state
cli                                     ; disable interrupts
push  OFFSET MyAESEventStructure        ; address of AES structure
call  CancelSleepAESProcessEvent
add   esp, 1*4                          ; adjust stack pointer
popfd                                   ; restore interrupt state
```

See Also

ScheduleSleepAESProcessEvent
ScheduleNoSleepAESProcessEvent, CancelNoSleepAESProcessEvent
ScheduleInterruptTimeCallBack, CancelInterruptTimeCallBack

CCheckHardwareInterrupt

[Non-blocking]

Syntax

long CCheckHardwareInterrupt (InterruptLevel) ;

Parameters

InterruptLevel
Specifies the Interrupt Level to be checked for a pending request.

Return Value

EAX is zero if no interrupt request is active for the specified interrupt level. A non-zero value indicates an interrupt request.

Requirements

Interrupts must be disabled on entry and will remain disabled.

Description

CCheckHardwareInterrupt determines if an interrupt request is currently being made to the Programmable Interrupt Controller (PIC) assigned to the indicated interrupt level. The PIC should normally have this level masked off while this call is made (interrupt will not be recorded by the PIC). This routine returns a value indicating the interrupt request status. A return value of zero indicates that the PIC has no interrupt request being made to it.

Example

```
push    IRQLevel                ;interrupt level (0-15)
call    CCheckHardwareInterrupt ;determine if active request
add     esp, 1 * 4              ;clean up stack
or      eax, eax                ;check status
jz      NoInterruptRequest
```

See Also

CDisableHardwareInterrupt, CEnableHardwareInterrupt
CDoEndOfInterrupt

CDisableHardwareInterrupt

[Non-Blocking]

Syntax

void CDisableHardwareInterrupt (InterruptLevel) ;

Parameters

InterruptLevel
Specifies the Interrupt Level to be masked off.

Return Value

None

Requirements

This routine can be called at either process or interrupt time. Interrupts must be disabled on entry and will remain disabled throughout this routine.

Description

This routine masks off the 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 adapter if possible. Disabling interrupts at the PIC is typically slower.

Example

```
DriverISR     proc

   movzx eax, BYTE PTR DriverConfiguration.CInterrupt
   push  eax
   call  CDisableHardwareInterrupt
   call  CDoEndOfInterrupt
    •
    •    (Service the adapter)
    •
   movzx eax, BYTE PTR DriverConfiguration.CInterrupt
   push  eax
   call  CEnableHardwareInterrupt
   ret

DriverISR     endp
```

See Also

DisableHardwareInterrupt
CEnableHardwareInterrupt, EnableHardwareInterrupt
CDoEndOfInterrupt, DoEndOfInterrupt

CDoEndOfInterrupt

[Non-Blocking]

Syntax

void CDoEndOfInterrupt (InterruptLevel) ;

Parameters

InterruptLevel
Specifies the Interrupt Level for the EOI command(s).

Return Value

None

Requirements

Interrupts must be disabled on entry and will remain disabled throughout this routine.

Description

This routine issues appropriate End of Interrupt (EOI) commands to the associated interrupt controller for the level indicated. 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 placing the code in the driver) allows flexibility when a driver runs on several platforms and ensures that this function is executed correctly.

Example

```
DriverISR     proc

   movzx eax, BYTE PTR DriverConfiguration.CInterrupt
   push  eax
   call  CDisableHardwareInterrupt
   call  CDoEndOfInterrupt
    •
    •    (Service the adapter)
    •
   movzx eax, BYTE PTR DriverConfiguration.CInterrupt
   push  eax
   call  CEnableHardwareInterrupt
   ret

DriverISR     endp
```

See Also

CEnableHardwareInterrupt, CDisableHardwareInterrupt
DoEndOfInterrupt, EnableHardwareInterrupt
DisableHardwareInterrupt

CEnableHardwareInterrupt

[Non-Blocking]

Syntax

void CEnableHardwareInterrupt (InterruptLevel) ;

Parameters

InterruptLevel
Specifies the Interrupt Level to be unmasked (enabled).

Return Value

None

Requirements

This routine can be called at either process or interrupt time. Interrupts must be disabled on entry and will remain disabled throughout this routine.

Description

The driver calls this routine to unmask (enable) the adapter's interrupt request line on the Programmable Interrupt Controller.

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

Example

```
DriverISR     proc

   movzx eax, BYTE PTR DriverConfiguration.CInterrupt
   push  eax
   call  CDisableHardwareInterrupt
   call  CDoEndOfInterrupt
    •
    •    (Service the adapter)
    •
   movzx eax, BYTE PTR DriverConfiguration.CInterrupt
   push  eax
   call  CEnableHardwareInterrupt
   ret

DriverISR     endp
```

See Also

CDisableHardwareInterrupt, CDoEndOfInterrupt
EnableHardwareInterrupt, DoEndOfInterrupt
DisableHardwareInterrupt

ClearHardwareInterrupt

[Non-Blocking]

Syntax

```
long   ClearHardwareInterrupt  (
                long  HardwareInterruptLevel,
                void  ( *InterruptProcedure ) ( void ) ;
```

Parameters

HardwareInterruptLevel
IRQ level of the hardware interrupt.

InterruptProcedure
Pointer to the driver's interrupt procedure.

Return Value

If *EAX* is equal to zero, the hardware interrupt was successfully removed. A non-zero value means the interrupt vector was not cleared because of invalid parameters or the vector was not found.

Requirements

This routine must only be called at process time (typically during initialization). Interrupts must be disabled on entry.

Description

ClearHardwareInterrupt releases a processor hardware interrupt previously allocated using *SetHardwareInterrupt*. This routine is called during the *DriverRemove* procedure when the driver is unloading or when the initialization procedure fails after an interrupt had been set.

Example

```
push  OFFSET DriverISR                    ; interrupt service routine
movzx eax, BYTE PTR DriverConfiguration.CInterrupt
push  eax                                 ; interrupt number
call  ClearHardwareInterrupt
add   esp, 2*4                            ; restore stack
or    eax, eax                            ; check for errors
jnz   ErrorReleasingInterrupt
```

See Also

SetHardwareInterrupt

CPSemaphore

[Blocking]

Syntax

void CPSemaphore (long SemaphoreNumber) ;

Parameters

SemaphoreNumber
Pointer to the semaphore.

Return Value

None

Requirements

This routine may only be called from a blocking process level. Interrupts may be in any state on entry and are preserved on return. However, during the call interrupts will be disabled.

Description

CPSemaphore is used to lock the real mode workspace when performing a real mode interrupt (such as an EISA BIOS call). For more information on how to use this procedure, refer to Appendix C.

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
```

See Also

GetRealModeWorkSpace
DoRealModeInterrupt
CVSemaphore

CRescheduleLast

[Blocking]

Syntax *void CRescheduleLast (void) ;*

Parameters None

Return Value None

Requirements This routine must only be called from the blocking process level as it will suspend the process and could change the machine state. Interrupts may be in any state on entry and that state is preserved on return. However, the interrupt state may be altered during execution of this procedure.

Description *CRescheduleLast* places the current task (the current driver process) last on the list of active tasks to be executed. Since the NetWare OS is non-preemptive, all driver processes normally run to completion. If a driver task requires too much execution time (i.e. retry loops), other scheduled processes may not execute in a timely manner. This routine can be used to temporarily release control so other scheduled tasks can execute (keeping vital OS processes working).

CRescheduleLast is normally used in conjunction with *AESSleepEvents* or in the driver initialization or remove procedures. The following example illustrates this call in a retry loop that attempts to redeliver a message to the OS after it has been placed on hold.

Example

```
HoldOffMessageDelivery:

   mov   HoldOffLoopCount, HOLDOFF_COUNT

HoldOffLoop:

   call  CRescheduleLast         ; Let other scheduled tasks execute
   dec   HoldOffLoopCount        ; we regain control here
   jnz   HoldOffLoop
     .
     .
     .
;Try to deliver the message to the OS

;If OS returns a "Holdoff Message" status again...

   jmp   HoldOffMessageDelivery
     .
     .
     .
```

See Also *DelayMyself*

CVSemaphore

[Non-Blocking]

Syntax

void CVSemaphore (long SemaphoreNumber) ;

Parameters

SemaphoreNumber
Pointer to the semaphore.

Return Value

None

Requirements

Interrupts may be in any state on entry and are preserved on return. However, during the call interrupts will be disabled.

Description

CVSemaphore clears a semaphore that was set with *CPSemaphore*.

Normally, *CVSemaphore* is used when the driver has finished performing a real mode interrupt (such as an EISA BIOS call) so that other processes can be allowed to use the workspace. For more information on how to use this procedure, refer to Appendix C.

Example

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

See Also

GetRealModeWorkSpace
CPSemaphore
DoRealModeInterrupt

DeAllocateMappedPages

Syntax

*void DeAllocateMappedPages (void *BufferPointer) ;*

Parameters

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

Return Value

None

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

See Also

AllocateMappedPages
Alloc, AllocBufferBelow16Meg
Free, FreeBufferBelow16Meg

DelayMyself

[Blocking]

Syntax

```
void   DelayMyself  (
            long  TimerTicks ,
            struct ResourceTagStructure *TimerResourceTag ) ;
```

Parameters

TimerTicks
Value indicating number of 1/18th second timer ticks to put this process to sleep (minimum time before return).

TimerResourceTag
Timer resource tag allocated by the driver during initialization.

Return Value

None

Requirements

This routine may only be called from a blocking process level. Interrupts may be in any state on entry and are preserved on return. However, interrupts might be enabled during this call.

Description

This routine delays the current process for the number of timer ticks specified by putting the current running process (the caller) to sleep. Return is made following expiration of the specified number of ticks. This routine is called to prevent a process from dominating the computer resources and preventing other vital processes from running. It also provides a specific minimum delay before the process is re-awakened, which may be helpful for tasks where some function will not complete for at least a specified period.

Example

```
push  TimerResourceTag            ;identify this driver
push  Ticks                       ;time to sleep
call  DelayMyself                 ;delay # ticks indicated
add   esp, 2 * 4                  ;clean up stack
```

See Also

CRescheduleLast, AllocateResourceTag

DeRegisterHardwareOptions

[Blocking]

Syntax

void DeRegisterHardwareOptions (
*struct IOConfigurationStructure *IOConfig) ;*

Parameters

IOConfig
Pointer to the *IOConfigurationStructure* that contains the adapter's hardware configuration to be deregistered.

Return Value

None

Requirements

This procedure must only be called from the blocking process level. Interrupts must be disabled.

Description

DeRegisterHardwareOptions releases the previously reserved hardware options specified in the adapter's *IOConfigurationStructure*. This routine is usually called from the driver's remove procedure.

Example

```
DriverRemove   proc

   CPush
   pushfd
   cli
    .
    .
    .
   push  OFFSET DriverConfiguration
   call  DeRegisterHardwareOptions
   add   esp, 1*4

   popfd
   CPop
   ret

DriverRemove   endp
```

See Also

RegisterHardwareOptions, ParseDriverParameters

DeRegisterServerCommDriver

[Blocking]

Syntax

long DeRegisterServerCommDriver (MSLResourceTag) ;

Parameters

MSLResourceTag
MSL resource tag allocated by the driver for *RegisterServer-CommDriver*.

Return Value

EAX is zero if the driver was successfully deregistered. A non-zero value indicates failure due to invalid parameters or the driver was not previously registered.

Requirements

This routine must only be called from a blocking process level. Interrupts must be disabled on entry.

Description

This procedure deregisters the driver from the Mirrored Server Link interface. *DeRegisterServerCommDriver* is normally called from the *DriverRemove* routine when the driver is unloaded. The SFT III operating system will be notified that the driver is no longer available for communications.

Example

```
push  MSLResourceTag
call  DeRegisterServerCommDriver     ;tell OS driver no longer available
add   esp, 1 * 4
```

See Also

RegisterServerCommDriver

DisableHardwareInterrupt

[Non-Blocking, Register-Based Routine]

On Entry

ECX specifies the interrupt level to be masked off (disabled).

On Return

EAX and *EDX* are destroyed; all other registers are preserved.

Requirements

This routine can be called at either process or interrupt time. Interrupts must be disabled on entry and will remain disabled throughout this routine.

Description

The driver calls this routine to mask off (disable) the adapter's interrupt request line on the Programmable Interrupt Controller (PIC). 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 adapter 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
   ret

DriverISR     endp
```

See Also

EnableHardwareInterrupt, DoEndOfInterrupt

DoEndOfInterrupt

[Non-Blocking, Register-Based Routine]

On Entry

ECX specifies the interrupt level for the EOI command(s).

On Return

EAX and *EDX* are destroyed; all other registers are preserved.

Requirements

This routine can be called at either process or interrupt time. Interrupts must be disabled on entry and will remain disabled throughout this routine.

Description

This routine issues the appropriate End of Interrupt (EOI) commands to the PIC (programmable interrupt controller) for the interrupt level specified. 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 placing the code in the driver) allows flexibility when a driver runs on several platforms and ensures that this function is executed correctly.

Example

```
DriverISR     proc

   mov   ecx, InterruptLevel
   call  DisableHardwareInterrupt
   call  DoEndOfInterrupt
    .
    .     (Service the adapter)
    .
   mov   ecx, InterruptLevel
   call  EnableHardwareInterrupt
   ret

DriverISR     endp
```

See Also

EnableHardwareInterrupt, DisableHardwareInterrupt

DoRealModeInterrupt

[Blocking]

Syntax

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

Parameters

InputParameters

Pointer to an InputParameter structure filled in with the register values required upon entry to the interrupt routine being called.

```
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

Pointer to an OutputParameter structure to be filled in by the interrupt routine with any register values returned.

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

Return Value

EAX is zero (0) if the interrupt vector is called successfully. A value of one (1) indicates the interrupt vector is no longer available because DOS has been removed. For some calls, certain OuputParameter values may also indicate success or failure.

Requirements

This routine must only be called from a blocking process level (normally during initialization). It may enable interrupts.

Description

DoRealModeInterrupt is used to perform real mode interrupts, such as BIOS and DOS interrupts. It will suspend server activity, switch to real mode, effect the interrupt, switch back to protected mode, and allow the server to resume activity.

EISA boards use *DoRealModeInterrupt* to perform an INT 15h BIOS call that obtains the board configuration. (For more information on how to use this procedure, refer to Appendix C)

Example

```
Note: The input parameter structure has already been initialized with the
      values required by the interrupt routine being executed.


push  OFFSET OutputParameters ; place pointer on stack
push  OFFSET InputParameters  ; place pointer on stack

call  DoRealModeInterrupt

add   esp, 2 * 4              ; clean up stack
or    eax, eax                ; check for error
jnz   RealModeInterruptError  ; handle error if necessary
```

See Also *CPSemaphore, CVSemaphore, GetRealModeWorkSpace*

EDXCallBackProcedure

[Non-Blocking, Register-Based Routine]

Syntax

call edx

On Entry

The registers must contain the message header values:

```
EAX   OS Parameter1
EBX   OS Parameter2
ECX   OS Parameter3 (message data length)
EDX   OS Parameter4 (pointer to the callback procedure)
ESI   OS Parameter5 (message destination pointer)
EDI   n/a
EBP   n/a
FLAGS
```

On Return

Assume all registers are destroyed.

Requirements

This routine is called from the interrupt level. Interrupts must be disabled on entry and are disabled on return.

Description

This procedure is called when the OS is notified of a received message (via the *ReceiveServerCommPointer* procedure) and a completion code of 1 is returned. A code of 1 indicates the driver must copy the message data from the adapter to system RAM and callback the procedure specified by EDX. This will return control to certain operating system receive procedures after the data has been copied.

On entry to this procedure, the registers must be set to the values contained in the original message header sent from the other server (with the exception of ECX and ESI which may have been modified during the *ReceiveServerCommPointer* routine).

Note: The *DriverSend* procedure may be called from within this callback routine, but will not enable interrupts.

Example

```
(See the ReceiveServerCommPointer example for an implementation of the
 EDX callback procedure)
```

See Also

ReceiveServerCommPointer, DriverISR

EnableHardwareInterrupt

[Non-Blocking, Register-Based Routine]

On Entry

ECX specifies the interrupt level to be unmasked (enabled).

On Return

EAX and *EDX* are destroyed; all other registers are preserved.

Requirements

This routine can be called at either process or interrupt time. Interrupts must be disabled on entry and will remain disabled throughout this routine.

Description

The driver calls this routine to unmask (enable) the adapter's interrupt request line on the Programmable Interrupt Controller (PIC).

Note: Novell recommends disabling and enabling interrupts at the adapter if possible. Controlling 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
   ret

DriverISR     endp
```

See Also

DisableHardwareInterrupt, DoEndOfInterrupt

Free

[Non-Blocking]

Syntax

*void Free (void *MemoryPtr) ;*

Parameters

MemoryPtr
Pointer to the allocated memory to be released.
(Must be memory previously allocated by the *Alloc* procedure.)

Return Value

None

Requirements

This routine may be called at either process or interrupt time. Interrupts may be in any state and will remain unchanged.

Description

Free returns a block of memory that was previously allocated by the driver using the *Alloc* routine. Drivers must free all allocated memory before exiting (typically during the *DriverRemove* procedure).

Example

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

See Also

Alloc
AllocBufferBelow16Meg, FreeBufferBelow16Meg
AllocateMappedPages, DeAllocateMappedPages

FreeBufferBelow16Meg

[Non-Blocking]

Syntax

*void FreeBufferBelow16Meg (void *MemoryPtr);*

Parameters

MemoryPtr
Pointer to the allocated memory to be released.
(Must be memory previously allocated by *AllocBufferBelow16Meg*.)

Return Value

None

Requirements

This routine may be called at either process or interrupt time. Interrupts may be in any state and will remain unchanged.

Description

FreeBufferBelow16Meg returns a block of memory that was previously allocated by the driver using the *AllocBufferBelow16Meg* routine. These routines are used by drivers that support 24-bit Bus Master or DMA adapters running in machines with more than 16 megabytes of memory.

Drivers must free all allocated memory before exiting (typically during the *DriverRemove* procedure).

Example

```
push  MyBelow16MegMemoryPtr      ; pointer to memory
call  FreeBufferBelow16Meg
add   esp, 1*4                   ; adjust stack pointer
```

See Also

AllocBufferBelow16Meg
Alloc, Free
AllocateMappedPages, DeAllocateMappedPages

FreeSemiPermMemory

[Non-Blocking]

Syntax

*void FreeSemiPermMemory (void *MemoryPtr);*

Parameters

MemoryPtr
Pointer to the allocated memory to be released.
(Must be memory previously allocated by *AllocSemiPermMemory*.)

Return Value

None

Requirements

This routine must only be called at process time. Interrupts may be in any state and will remain unchanged.

Description

FreeSemiPermMemory returns a block of memory that was previously allocated by the driver using the *AllocSemiPermMemory* routine. Drivers must free all allocated memory before exiting (typically during the *DriverRemove* procedure).

Example

```
push  MyMemoryPtr                ; pointer to memory
call  FreeSemiPermMemory
add   esp, 1*4                   ; adjust stack pointer
```

See Also

AllocSemiPermMemory
Alloc, Free
AllocateMappedPages, DeAllocateMappedPages

GetCurrentTime

[Non-Blocking]

Syntax

long GetCurrentTime (void) ;

Parameters

None

Return Value

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

Requirements

None

Description

GetCurrentTime can be used to determine the elapsed time (in ticks) for driver-related events (such as timeout checks). The current time value minus the value returned at the start of an operation is the elapsed time in 1/18th second clock ticks. This timer requires more than 7 years to roll over, allowing it to be used for elapsed time comparisons.

Example

```
call  GetCurrentTime                ; get transmit start time for
mov   TransmitStartTime, eax        ; timeout checking
```

GetHardwareBusType

[Non-Blocking]

Syntax

long GetHardwareBusType (void) ;

Parameters

None

Return Value

EAX contains a value indicating the bus type.

```
0 = ISA  (Industry Standard Architecture)
1 = MCA  (Micro-Channel Architecture)
2 = EISA (Extended Industry Standard Architecture)
```

Requirements

This routine can be called at either process or interrupt time. Interrupts may be in any state on entry and will remain unchanged.

Description

GetHardwareBusType returns a value indicating the processor bus type.

This routine would allow a single driver to support boards for different bus types, which, following initialization and configuration, appear identical to the driver.

Example

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

GetNextPacketPointer

[Non-Blocking, Register-Based Routine]

Syntax

call [GetNextPacketPointer]

On Entry

None (The driver should be ready to have its *BuildSend* routine called)

On Return

Assume all registers are destroyed. Upon return, the driver should send the built packet.

Requirements

This routine is called from interrupt level. Interrupts must be disabled on entry.

Description

GetNextPacketPointer is a global variable defined by the OS. It contains a pointer to the current OS routine used to obtain any messages queued for transmission.

After the driver receives an acknowlegement from the other server for a message (or group of messages) previously sent, it notifies the OS of the acknowledgements via *SendServerCommCompletedPointer*. The driver must then check if the OS queued up any messages while it was busy transmitting that last message.

The OS indicates the size of the next queued message (excluding headers) using the *PacketSizeNowAvailable* variable. If no messages are queued for transmission, this value is negative. (A value of zero indicates a message header only with no message data.) The size of the message will always be less than or equal to the maximum data size the MSL driver is capable of sending.

If there are messages queued, the driver must make an indirect call to this routine. Calling this procedure initiates a possible multimessage building sequence. During the *GetNextPacket* routine, the *DriverBuild-Send* procedure is called repeatedly to build the multimessage packet. *GetNextPacket* will stop calling the *DriverBuildSend* routine only when the driver indicates, through the value in *PacketSizeDriverCan-NowHandle*, that it has no more room for additional messages, or when the OS has no more messages to send. (See the flow chart and explanation of building a multimessage packet in Chapter 4, under the *DriverBuildSend* procedure)

See Also

DriverBuildSend, DriverISR
GetNextPacketPointer global variable (defined in Chapter 3)

Example

```
DriverISR   proc

   .
   .
   .

;*************************************************************************
;* Acknowledgement Received                                              *
;*************************************************************************

ISRAckReceived:

    cmp     MessageInProgress, TRUE                 ;validate ack
    jne     CheckAdapterStatus                      ;


    ;*********************************************************************
    ;* Cancel Message TimeOut Sequence                                   *
    ;*********************************************************************

    mov     MessageInProgress, FALSE                ;clear flag
    mov     TimeOutEvent.MessageTimeOutTime, 0      ;stop meessage timer


    ;*********************************************************************
    ;* Notify OS of the acknowledgement(s)                               *
    ;*********************************************************************

    mov     ebp, TxPacketMessageCount               ;get # of messages sent
    add     ReceiveAckCount, ebp                    ;update statistics counter
    call    [SendServerCommCompletedPointer]        ;notify OS of ACKs
                                                    ;(use indirect call)

    ;*********************************************************************
    ;* Transmit any queued messages in possible multi-message packet     *
    ;*********************************************************************

    mov     PacketSizeDriverCanNowHandle, MAX_PACKET_SIZE  ;size MSL can send
    cmp     PacketSizeNowAvailable, MAX_PACKET_SIZE        ;anything queued?
    ja      CheckAdapterStatus                             ;jump if not


    ;(Set up any variables for the DriverBuildSend routine:  Typically, set
    ; ptr to next transmit buffer, and reset any message counters to zero.)


    mov     TxPacketMessageCount, 0
    call    [GetNextPacketPointer]                  ;start BuildSend sequence
                                                    ; (using indirect call)
    mov     PacketSizeDriverCanNowHandle, -1        ;semaphore no more sends


    ;At this point one or messages have been built in the packet via repeated
    ;calls to the DriverBuildSend Routine and are ready for transmission.


    ;*********************************************************************
    ;* Transmit the message packet and start a transmit timeout sequence *
    ;*********************************************************************

    call    TransmitMessagePacket                   ;(see template)

    inc     TransmitBurstPacketCount                ;update statistics counter
    jmp     CheckAdapterStatus
```

GetProcessorSpeedRating

[Non-Blocking]

Syntax

long GetProcessorSpeedRating (void) ;

Parameters

None

Return Value

EAX contains a value representing the relative processor speed of the machine. A value of zero indicates the routine failed to determine the processor speed.

Requirements

This routine may be called at either process or interrupt time. Interrupts can be in any state on entry and will not be changed during the routine.

Description

GetProcessorSpeedRating is used to determine the relative processor speed. 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
```

GetRealModeWorkSpace

[Non-Blocking]

Syntax

```
void   GetRealModeWorkSpace  (
            struct SemaphoreStructure   *WorkSpaceSemaphore ,
            long    *WorkSpaceProtectedModeAddress ,
            word    *WorkSpaceRealModeSegment ,
            word    *WorkSpaceRealModeOffset ,
            long    *WorkSpaceSize ) ;
```

Parameters

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 below.

```
WorkSpaceSemaphore             dd 0
WorkSpaceProtectedModeAddress  dd 0
WorkSpaceRealModeSegment       dw 0
WorkSpaceRealModeOffset        dw 0
WorkSpaceSize                  dd 0
```

WorkSpaceSemaphore
Pointer to the operating system semaphore structure.

WorkSpaceProtectedModeAddress
32-bit logical address of the workspace.

WorkSpaceRealModeSegment
Real mode segment of the workspace.

WorkSpaceRealModeOffset
Real mode offset in the workspace segment.

WorkSpaceSize
Size of the workspace.

Return Value

None (all values are returned through the parameters)

Requirements

This routine can be called at either process or interrupt time. Interrupts can be in any state and will remain unchanged.

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. (For more information on how to use this procedure, refer to Appendix C)

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].IntNumber, 15h                    ; interrupt number

push  OFFSET OutputParms                      ; pointer to output regs
push  OFFSET InputParms                       ; pointer to input regs
call  DoRealModeInterrupt
add   esp, 2*4                                ; clear up stack
or    eax, eax                                ; error check
jnz   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

[Non-Blocking]

Syntax

long GetServerPhysicalOffset (void) ;

Parameters

None

Return Value

EAX contains a 32-bit physical address of the operating system's logical address 0.

Requirements

This routine may be called at either process or interrupt time. Interrupts may be in any state on entry and will remain unchanged.

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.

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. For example:

PhysicalAddress = LogicalAddress + GetServerPhysicalOffset ( ) ;
LogicalAddress = PhysicalAddress – GetServerPhysicalOffset ( ) ;

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
mov   ServerPhysicalOffset, eax
```

GetSharedMemoryLinearAddress

[Non-Blocking]

Syntax

```
long   GetSharedMemoryLinearAddress  (
            long  SharedMemoryPhysicalAddress,
            long  Size ) );
```

Parameters

SharedMemoryPhysicalAddress
32-bit physical address.

Size
Size of the shared memory space.

Return Value

EAX contains a logical 32-bit address relative to the operating system's assignment of address 0h.

Requirements

Description

GetSharedMemoryLinearAddress returns the 32-bit logical address assigned to the physical address passed to the routine.

Note: Although the MLID may get this address, it may not be able to access it. For example, if the driver used this call to access shared RAM prior to registering hardware options, a page-fault exception would occur.

Example

```
push  SharedRAMSize
push  SharedRAMPhysicalAddress
call  GetSharedMemoryLinearAddress  ;EAX = address to use
add   esp, 2*4                      ;Restore stack
mov   SharedRAMLogicalAddress, EAX
```

OutputToScreen

[Non-Blocking]

Syntax

```
long   OutputToScreen  (
            struct ScreenStruct  *ScreenHandle ,
            char  *ControlString ,
            args... ) ;
```

Parameters

ScreenHandle
ScreenHandle that NetWare passed on the stack to the driver's initialization routine at load time.

ControlString
Pointer to a null-terminated ASCII string (similar to the standard *printf* string).

args...
Procedure can take a variable number of arguments as required by the control string format specifiers.

Return Value

EAX is zero if successful. A non-zero value indicates an error has occurred.

Requirements

This routine must only be called during *DriverInitialize*, since the driver's *ScreenHandle* is valid only during the initialization routine.

Description

OutputToScreen is used to display a driver error message on the server console screen using standard printf formatting.

Drivers should not display non-vital messages and should limit the number of lines output to the screen for essential messages. Displaying unneeded output will cause important information to scroll off the screen.

ControlString can be embedded with returns, line feeds, bells, tabs, and % specifiers (except floating point). However, if strings contain embedded substrings, numbers and control information, they must be limited in length to a maximum of 200 characters. 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.

Example

```
DriverInitialize  proc

  CPush
  mov    ebp, esp
  pushfd
  cli
   .
   .
   .

PrintErrorMessage:

  push   OFFSET MyErrorMessage                  ; push offset to message
  push   [ebp + Parm1]                          ; screen handle
  call   OutputToScreen
  add    esp, 2*4                               ; restore stack
   .
   .
   .

DriverInitialize  proc
```

See Also *QueueSystemAlert*

ParseDriverParameters

[Blocking]

Syntax

```
long  ParseDriverParameters (
        struct IOConfigurationStructure *IOConfig,
        void   unused1,
        struct AdapterOptionStructure *AdapterOptions,
        void   unused2,
        void   unused3,
        long   NeedsBitMap,
        byte  *CommandLine,
        struct ScreenStruct *ScreenHandle );
```

Parameters

**IOConfig*

Pointer to adapter's *IOConfigurationStructure*. The structure must be initialized and contain a valid IOResourceTag. (See Chapter 3 for a description of the *IOConfigurationStructure*)

**AdapterOptions*

Pointer to the driver's *AdapterOptionStructure*. A driver typically maintains one option structure, although multiple structures may be used if the driver supports more than one adapter type requiring different parameters.

The *AdapterOptionStructure* is defined as follows:

```
AdapterOptionStructure    struc
  IOSlot          dd  ?     ;MCA or EISA slot #
  IOPort0         dd  ?     ;I/O port base
  IOLength0       dd  ?     ;range (# ports)
  IOPort1         dd  ?     ;2nd I/O port base
  IOLength1       dd  ?     ;range (# ports)
  MemoryDecode0   dd  ?     ;memory (SRAM/EPROM)
  MemoryLength0   dd  ?     ;range (paragraphs)
  MemoryDecode1   dd  ?     ;2nd memory base
  MemoryLength1   dd  ?     ;range (paragraphs)
  Interrupt0      dd  ?     ;Interrupt #
  Interrupt1      dd  ?     ;2nd Int #
  DMA0            dd  ?     ;DMA channel
  DMA1            dd  ?     ;2nd DMA channel
AdapterOptionStructure    ends
```

Each field in the above structure is a pointer to a table of valid options for that parameter. If a parameter is not required or used by the driver/adapter, set the field to zero (a null pointer).

Each option table must begin with a dword indicating the number of options in the list. The options listed in the tables represent valid values that may be selected from the command line. The default value (if none is specified) is the first unused value in the table.

Parameters (continued)

A sample option list follows:

```
PortOptionTable   dd    4       ;number of port options
                  dd    340h    ;first (default) port
                  dd    344h    ;second possible port
                  dd    320h    ;third possible port
                  dd    324h    ;last possible port
```

NeedsBitMap

A bit map (dword value) telling *ParseDriverParameters* which hardware options the driver requires, as follows:

```
NeedsIOSlotBit          equ   0001h
NeedsIOPort0Bit         equ   0002h
NeedsIOLength0Bit       equ   0004h
NeedsIOPort1Bit         equ   0008h
NeedsIOLength1Bit       equ   0010h
NeedsMemoryDecode0Bit   equ   0020h
NeedsMemoryLength0Bit   equ   0040h
NeedsMemoryDecode1Bit   equ   0080h
NeedsMemoryLength1Bit   equ   0100h
NeedsInterrupt0Bit      equ   0200h
NeedsInterrupt1Bit      equ   0400h
NeedsDMA0Bit            equ   0800h
NeedsDMA1Bit            equ   1000h
```

Note: It is invalid to indicate that an entry is required by setting the associated bit in the *NeedsBitMap* while having a null pointer in the *AdapterOptionStructure* or having the number of options in an option table indicated as zero.

CommandLine

Pointer to command line passed to the driver's Initialize routine on the stack at load time.

ScreenHandle

Pointer to the driver's screen display. NetWare also passed this value to the driver's initialization routine on the stack at load time.

(unused parameters)

Unused parameters should be set to zero.

Return Value

EAX is zero if successful. A non-zero value indicates conflict with existing hardware or bad command line parameters.

Requirements

This routine may only be called from a blocking process level. In addition, it may only be used during *DriverInitialize*, since the driver's *ScreenHandle* is valid only during the initialization routine.

Description

ParseDriverParameters allows a driver's initialization routine to obtain hardware configuration information from the load command line. This may include the slot number, I/O ports and ranges, memory decode addresses and lengths, interrupts, and/or DMA addresses. The information obtained from the command line is placed in the appropriate fields of the driver's *IOConfigurationStructure*.

For example, a load command could contain the following specifications:

```
load drivername port=300:20, int=3
```

In this case, the adapter will occupy I/O ports 300h to 31Fh and use interrupt 3. The load command line keywords associated with each field of the *IOConfigurationStructure* are listed in Appendix A

ParseDriverParameters fills in the *IOConfigurationStructure* associated with an adapter utilizing tables provided by the driver, the command line parameters, and operator input. The following describes this process:

The driver specifies which hardware configuration options the adapter needs with the *NeedsBitMap*. Using this mask as a guide, *ParseDriverParameters* collects the required information from the command line, validates the parameters against the values provided via the *AdapterOptionStructure*, and fills out the appropriate fields of the *IOConfigurationStructure*.

If the *NeedsBitMap* requires data for a particular option and *ParseDriverParameters* cannot find the data on the command line, it will prompt the console operator for the data, showing as a default the first unused entry in the option table pointed to by the associated field in the *AdapterOptionStructure*.

Once all required fields of the *IOConfigurationStructure* have been filled in, the hardware configuration must be registered with the OS using the NetWare routine, *RegisterHardwareOptions*. This routine checks for conflicts with existing hardware and reserves the specified file server options for the adapter's use (if no conflicts exist).

Note: Refer to Appendix C, "Obtaining Configuration Information," for detailed instructions on determining the hardware configuration for EISA and MCA machines.

Custom Command Line Keywords

The driver may implement additional command line keywords that it alone recognizes. If the driver defines custom keywords, it must parse them from the command line itself. The driver should *not* adjust the the pointer to the command line or delete the custom keywords from the command line text, since the *ParseDriverParameters* routine will simply ignore the additional parameters.

Example

```
DriverInitialize   proc

   CPush
   mov     ebp, esp
   pushfd
   cli
   .
   .
   .
   push    [ebp+Parm1]                          ;Screen Handle
   push    [ebp+Parm2]                          ;Command Line pointer
   push    NeedsIOPort0Bit + NeedsInterrupt0Bit ;need I/O port & interrupt
   push    0                                    ;unused
   push    0                                    ;unused
   push    OFFSET AdapterOptions                ;card options template
   push    0                                    ;unused
   push    OFFSET DriverConfiguration           ;IOConfig structure
   call    ParseDriverParameters                ;fill out IOConfig Struct
   add     esp, 8 * 4                           ;clean up stack
   or      eax, eax                             ;check for errors
   jnz     ErrorParsingCommandLine              ;jump on error


   push    0                                    ;unused
   push    OFFSET DriverConfiguration           ;IOConfig structure
   call    RegisterHardwareOptions              ;register configuration
   add     esp, 2 * 4                           ;clean up stack
   or      eax, eax                             ;check for errors
   jnz     ErrorRegisteringHardware             ;jump on error
   .
   .
   .
```

See Also *AdapterOptionStructure, IOConfigurationStructure*
RegisterHardwareOptions, DeRegisterHardwareOptions
Appendix C, "Obtaining Configuration Information"
ReadEISAConfig

QueueSystemAlert

[Non-Blocking]

Syntax

```
long  QueueSystemAlert (
          long  TargetStation ,      long  TargetNotificationBits ,
          long  ErrorLocus ,         long  ErrorClass ,
          long  ErrorCode ,          long  ErrorSeverity ,
          byte *ControlString ,      args . . . ) ;
```

Parameters

TargetStation

Connection number of the affected station. This is normally set to zero meaning that no single station is affected. Supply a zero for the console.

TargetNotificationBits: Destinations of the notification.

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

ErrorLocus: Locus of the error.

LOCUS_UNKNOWN	00h
LOCUS_LANBOARDS	04h

ErrorClass: Class of the error.

CLASS_UNKNOWN	00h
CLASS_TEMP_SITUATION	02h
CLASS_HARDWARE_ERROR	05h
CLASS_BAD_FORMAT	09h
CLASS_MEDIA_FAILURE	11h
CLASS_CONFIGURATION_ERROR	15h
CLASS_DISK_INFORMATION	18h

ErrorCode: Error codes for the system log.

OK	00h
ERR_HARD_FAILURE	FFh

ErrorSeverity: Severity of the error.

SEVERITY_INFORMATIONAL	00h
SEVERITY_WARNING	01h
SEVERITY_RECOVERABLE	02h
SEVERITY_CRITICAL	03h
SEVERITY_FATAL	04h
SEVERITY_OPERATION_ABORTED	05h

Parameters (continued)

ControlString

Pointer to a null-terminated control string similar to the standard *printf* string used in the output routine. The string can include embedded returns, linefeeds, tabs, bells, and % format specifiers (except floating point).

args...

The routine can take a variable number of arguments as required by the control string format specifiers.

Return Value

EAX is 0 if successful. A value of 1 means the alert was not available.

Requirements

This routine may be called at either process or interrupt time. Interrupts may be in any state on entry and will remain unchanged.

Description

QueueSystemAlert provides system notification of driver hardware or software problems during regular operation of the board *(at times other than during the driver initialization procedure)*.

Example

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


   movzx eax, [ebx].CDriverBoardNumber    ; argument: board number
   push  eax
   push  OFFSET TransmitTimeoutMessage    ; ControlString
   push  SEVERITY_RECOVERABLE             ; ErrorSeverity
   push  OK                               ; ErrorCode
   push  CLASS_HARDWARE_ERROR             ; ErrorClass
   push  LOCUS_LANBOARDS                  ; ErrorLocus
   push  NOTIFY_ERROR_LOG_BIT OR NOTIFY_CONSOLE_BIT
   mov   eax, 0
   push  eax                              ; station #, not used
   call  QueueSystemAlert
   add   esp, 8*4                         ; clean up stack
```

See Also

OutputToScreen

ReadEISAConfig

[Register-Based Routine]

On Entry

CH=Block *CL*=Slot

On Return

EAX contains:

```
00h = successful (zero flag is also set)
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 points to the buffer containing the configuration.
EDX and *EDI* are destroyed.

Requirements

This routine may only be called at process time, normally during initialization. Interrupts may be in any state on entry and that state is preserved on return. However, interrupts might be enabled during the execution of this procedure.

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.

Example

```
DriverInitialize  proc
    .
    .
    .
   movzx ecx, DriverConfiguration.CSlot   ; ch = block 0, cl = slot

ReadConfigBlockLoop:

   call  ReadEISAConfig                   ; get config block
   jnz   ErrorReadingEISAConfig           ; check for errors
   inc   ch                               ; set up for next block
   test  BYTE PTR [esi+n], Valid_Data     ; does buffer contain desired data
   jz    ReadConfigBlockLoop              ; try next config block
```

See Also

ParseDriverParameters

ReadRoutine

[Blocking]

Syntax

```
long   ( *ReadRoutine ) (
             long   CustomFileHandle ,
             long   CustomDataOffset ,
             long  *CustomDataDestination ,
             long   CustomDataSize ) ;
```

Parameters

CustomFileHandle
Module handle of the .MSL file. This value is passed to the driver's initialization routine as the *LoadableModuleFileHandle* parameter.

CustomDataOffset
Starting offset in the file. This value is the *CustomDataOffset* parameter passed to the driver's initialization routine.

CustomDataDestination
Location of a driver allocated buffer that the *ReadRoutine* should use as the destination for the custom data file.

CustomDataSize
Amount of custom data (in bytes) to read. This value is the *CustomDataSize* parameter passed to the driver's initialization routine.

Return Value

EAX is zero if successful. A non-zero value indicates failure.

Requirements

This routine may be called only during initialization. Interrupts may be enabled on return.

Description

Some drivers may require custom firmware or data to download to the adapter during initialization. The *ReadRoutine* allows drivers to read custom data or firmware into system memory during initialization.

The entry point of the *ReadRoutine* is not exported by the operating system. A pointer to the routine is passed on the stack to the driver during initialization and must be called indirectly. The only place it is valid is in the initialization routine.

With the exception of the *CustomDataDestination*, Netware passes all the parameters required by this routine on the stack to the driver's initialization routine. Before this routine is called, the driver must allocate a buffer that the *ReadRoutine* uses as the destination for the custom data file.

The NetWare linker actually appends the custom data file to the MSL module at link time. When the driver is loaded, NetWare loads only the driver's code, leaving the file open for the driver to handle its custom data. To attach a custom file to the driver module, use the CUSTOM keyword in the driver's linker definition file followed by the name of the custom file.

Note: The following example assumes that the custom file has been attached to the driver module as described above.

Example

```
DriverInitialize  proc

   CPush
   mov   ebp, esp
   pushfd
   cli
    •
    •
    •
   push  MemoryRTag                          ; push memory resource tag
   mov   eax, [ebp + Parm8]                  ; get CustomDataSize from stack
   push  eax                                 ; push size
   call  Alloc                               ; allocate memory
   add   esp, 2*4                            ; clean up stack
   or    eax, eax                            ; did we get it?
   jz    ErrorGettingMemory                  ; error exit if not
   mov   FirmwareBufferPtr, eax              ; save firmware buffer

   mov   eax, [ebp + Parm8]                  ; CustomDataSize
   push  eax                                 ;
   push  FirmwareBufferPtr                   ; CustomDataDestination
   mov   eax, [ebp + Parm7]                  ; CustomDataOffset
   push  eax                                 ;
   mov   eax, [ebp + Parm5]                  ; LoadableModuleFileHandle
   push  eax                                 ;

   mov   ebx, [ebp + Parm6]                  ; ReadRoutine ptr
   call  ebx                                 ; call read routine

   cli                                       ; clear interrupts
   add   esp, 4*4                            ; adjust the stack
   or    eax, eax                            ; check for read errors
   jnz   ReadError                           ; jump if errors
    •
    •
    •
```

See Also *DriverInitialize, AllocateResourceTag*
Alloc, AllocateMappedPages, AllocBufferBelow16Meg

ReceiveServerCommPointer

[Non-blocking, Register-based Routine]

Syntax

call [ReceiveServerCommPointer]

On Entry

The registers must be set to the values in the message header of the received message:

```
EAX   OS Parameter1
EBX   OS Parameter2
ECX   OS Parameter3 (data length, may be zero)
EDX   OS Parameter4
ESI   OS Parameter5 (data pointer, if ECX is non-zero)
EDI   OS Parameter6
```

On Return

```
EAX   completion code (defined below)
EBX   assume EBX is destroyed
ECX   data length (may be zero)
EDX   callback address if AL=1, otherwise assume EDX
      is destroyed
ESI   data destination pointer (if ECX is non-zero)
EDI   (not needed after this call)
```

Requirements

This routine is called from interrupt level. Interrupts must be disabled on entry and will remain disabled.

Description

ReceiveServerCommPointer is a global variable defined by the OS. It contains a pointer to the current procedure used to notify the OS when a message is received from the other server. Before making the indirect call to this routine, the driver must set the registers to the values stored in the message header sent from the other server. This routine then returns a completion code indicating to the driver what action to take with the message data.

This routine must be called for each message received from the other server. If there are multiple messages in a packet, it is the driver's responsibility to deliver the individual messages.

The OS may call *DriverSend* during this routine; therefore, the driver must be capable of sending a packet at this point.

Completion Codes

The completion code (CCode) returned by this routine will be a number between 0 and 4 indicating what action to take with the message data. The CCode values are described in the following section.

Note: The OS may modify the ECX (message size) and ESI (message destination) registers during this call, effectively bypassing or ignoring the data. The modified values must be used to copy the message if required, however the driver should save the original message size in order to adjust the pointers into the received packet.

CCode = 0 (OK: copy message)

The driver should copy ECX bytes of the message data from the adapter to the destination in system RAM specified by ESI.

Note: ECX and ESI may have been modified by the routine. The new values returned by this routine must be used for the data copy.

CCode = 1 (OK: copy message and callback)

The driver should copy ECX bytes of the message data from the adapter to the destination in system RAM specified by ESI.

Note: ECX and ESI may have been modified by this routine. The values returned from this routine must be used for the data copy.

After copying the data, the driver should callback the receive handler whose address is specified in EDX. Prior to making the callback, the registers must be restored to the original message header values with the exception of ECX and ESI, which should contain the values returned by this routine.

CCode = 2 (Holdoff message)

This code signals the driver to place the message on hold for redelivery at a later time. The driver may either send a request asking the other server to resend the packet or save the packet and attempt to deliver the message at a later time. Care must be taken to ensure that messages are not delivered to the OS twice.

The hold state is used by the operating system to throttle the incoming packets. Since the OS requires messages to be delivered in sequence, any messages received following the holdoff state cannot be delivered until the heldoff message is successfully delivered.

Note: The operating system needs to run before it will be able to accept the heldoff message. An immediate attempt to redeliver the message without relinquishing control will be fruitless. Redelivery can be accomplished by setting up asynchronous or interrupt time callback events that relinquish control, then trigger an attempt to redeliver the message. (Refer to the *DriverHoldOff* and *DriverInt-HoldOff* procedure descriptions in Chapter 4, "MSL Driver Procedures," for more information on implementing redelivery of heldoff messages.)

CCode = 3 (Holdoff message)

Same as CCode 2.

CCode = 4 (Ignore)

The driver should ignore this message and continue on to the next message.

Acknowledging Messages

Once the driver has successfully delivered the message packet data to the OS, it should send an acknowledgement to the other server. For efficiency, only one acknowledgement should be sent to ACK all messages in the packet.

To minimize latency, the driver can send the acknowledgement before delivering the messages to the OS (as in the example template in Appendix E). The driver must then guarantee that the messages are delivered, otherwise the server states will diverge. The driver must also be able to receive a second message packet from the other server.

If an acknowledged message is placed on hold and a second message packet is received, the MSL driver should not acknowledge the second message packet. Instead, it should begin sending *holdoff notifications* once every clock tick, to prevent the other server from inadvertently timing out on any message packets that have already been sent, but have not yet received an acknowledgement from this server. This also has the affect of stopping the flow of messages from the other server since its driver will normally indicate it can send another message (via the *PacketSizeDriverCanNowHandle* variable), only when it receives the acknowledgement for the last message packet transmitted.

The mirrored server drivers must be capable of receiving acknowledgements (as well as holdoff and emergency notifications) during a message holdoff state. This is required to prevent a possible “deadlock” situation in which both servers are waiting on something from the other server in order to clear the holdoff state.

The MSL driver must also handle error reporting differently when a message is placed on hold after it has been acknowledged. Refer to the *ServerCommDriverError* procedure later in this chapter for information on error handling.

Example

```
DriverISR   proc

    .
    .
    .

;*************************************************************************
;* Message Packet Received                                               *
;*************************************************************************

ISRMessagePacketReceived:

    cmp     HoldStateFlag, 0                ;if last message packet held...
    jne     ISRHoldOffMessage               ;...hold this one

    call    TransmitAcknowledgement         ;else transmit acknowledgement


    (point to the first message header in receive buffer)



ISRProcessMessage:


    (read in message header then set registers to these values)


    mov     eax, MessageHeader.EaxParameter
    mov     ebx, MessageHeader.EbxParameter
    mov     ecx, MessageHeader.EcxParameter
    mov     edx, MessageHeader.EdxParameter
    mov     esi, MessageHeader.EsiParameter
    mov     edi, MessageHeader.EdiParameter

    call    [ReceiveServerCommPointer]              ;inform OS of message

    mov     NewEsiParameter, esi
    mov     NewEcxParameter, ecx

    cmp     al, 0
    je      ISRCopyMessage

    cmp     al, 1
    je      ISRCopyMessageAndCallBackOS

    cmp     al, 4
    jb      ISRHoldOffMessage



ISRProcessNextMessage:

    dec     PacketHeader.MSLMessageCount            ;1 less message to process
    jz      ISRReceiveMessageDone                   ;jump if no more


    (point to next message header in receive buffer)


    jmp     ISRProcessMessage                       ;hand next msg to OS
```

Example

```
ISRCopyMessage:

    or      ecx, ecx                                ;any data to copy?
    jz      ISRProcessNextMessage                   ;if not, process next message

    (copy message data to OS memory:  NewEcx=size  NewEsi=addr)

    jmp     ISRProcessNextMessage                   ;process next message



ISRCopyMessageAndCallBackOS:

    or      ecx, ecx                                ;any data to copy?
    jz      ISRCallBackOS                           ;if not, skip data copy

    (copy message data to OS memory:  NewEcx=size  NewEsi=addr)

ISRCallBackOS:

    mov     eax, MessageHeader.EaxParameter         ;original eax parameter
    mov     ebx, MessageHeader.EbxParameter         ;original ebx parameter
    mov     ecx, NewEcxParameter                    ;use new ecx parameter
    mov     esi, NewEsiParameter                    ;use new esi parameter

    call    edx                                     ;call to OS

    jmp     ISRProcessNextMessage                   ;process next message



ISRHoldOffMessage:

;*************************************************************************
;*                                                                       *
;*  This example assumes the following:                                  *
;*                                                                       *
;*    1.  The received packet can not be left on the adapter after       *
;*        reading the message header.                                    *
;*    2.  The adapter has a receive buffer for more than one             *
;*        maximum size packet.                                           *
;*                                                                       *
;*************************************************************************

    (copy the message packet into the receive/hold buffer)
    (update all receive/hold buffer pointers)

    inc     HoldStateFlag                           ;indicate hold state
    cmp     HoldStateFlag, 2                        ;check for previous holds
    je      ISRReceiveMessageDone                   ;if so we're done here

    call    TransmitHoldNotification                ;else notify other server
                                                    ;of hold state

    ;*********************************************************************
    ;* Setup callbacks for message redelivery attempts                   *
    ;*********************************************************************

    push    OFFSET HoldOffEvent
    call    ScheduleSleepAESProcessEvent
    add     esp, 1 * 4

    mov     edx, OFFSET IntHoldOffEvent
    call    ScheduleInterruptTimeCallBack

    jmp     ISRReceiveMessageDone
```

RegisterForEventNotification

Syntax

```
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 with an *EventSignature* obtained by the driver for event notification. (see *AllocateResourceTag*)

EventType
Type of event for which notification is desired.

Priority
Order in which registered callback routines will be called.

WarnProcedure
Pointer to a callback routine which will be called when EventCheck is called.

OutputRoutine
Used to warn the user against a particular event.

ControlString
Pointer to a null-terminated string similar to a standard printf control string that will be used in the *OutputRoutine*. The string can include embedded returns, linefeeds, tabs, bells, and % format specifiers (except floating point).

args...
The *OutputRoutine* can take a variable number of arguments as required by the *ControlString* format specifiers.

ReportProcedure
Pointer to a callback routine that is called when EventReport is called.

Parameter(s)
32-bit value that is defined according to the event type.

Return Value

EAX contains an *EventID* that should be used when calling *UnRegisterEventNotification*. A value of zero indicates failure to register the event notification.

Description

RegisterForEventNotification is called at initialization to register a driver-defined routine for callback if a particular event type occurs. For example, the driver can register a routine so that it can be notified if the server is going to exit to DOS. This will give the driver a chance to service the physical board, to cancel any AES or timer events, and to allow bus master devices to return pre-allocated resources and shutdown the adapter before the OS exits to DOS. This is especially important for DMA or bus master devices that need to be shutdown to prevent them from writing to memory after DOS gets control.

This procedure will add the specified routines to the event list when an event is reported. The *WarningProcedure* will be called when an *EventCheck* is called by the operating system, and the *ReportProcedure* will be called when an *EventReport* is called by the operating system. These routines will be called according to priority. The parameter passed in when the event is reported will be passed to the *Warning-Procedure* or *ReportProcedure* when it is called.

When the type of event (defined by *EventType*) occurs, the operating system calls the specified callback routine. The types of defined events are listed below:

```
EVENT_DOWN_SERVER              04h
```
The parameter is undefined. The warn routine and the report routine will be called before the server is shut down.

```
EVENT_CHANGE_TO_REAL_MODE      05h
```
The parameter is undefined. The report routine will be called before the server changes to real mode and must not go to sleep.

```
EVENT_RETURN_FROM_REAL_MODE    06h
```
The parameter is undefined. The report routine will be called after the server returns from DOS and must not go to sleep.

```
EVENT_EXIT_TO_DOS              07h
```
The parameter is undefined. The report routine will be called before the server exits to DOS.

The order in which the callback routines will be called is determined by the priority parameter with the priorities being notified first. The available priorities are listed below:

```
EVENT_PRIORITY_OS           00h
EVENT_PRIORITY_APPLICATION  20h
EVENT_PRIORITY_DEVICE       40h
```

When the *WarnProcedure* is called, it is passed a *Parameter* and a pointer to an *OutputRoutine* that the driver should use to warn the user against the occurrence of a particular event. Nulls may be passed to the routine.

The *ReportProcedure* is 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
```

RegisterHardwareOptions

[Non-Blocking]

Syntax

```
long   RegisterHardwareOptions  (
            struct IOConfigurationStructure  *IOConfig ,
            void  unused ) ;
```

Parameters

IOConfig
Pointer to the adapter's completed IOConfiguration structure.

unused
This parameter is used only by LAN drivers and should be set to zero for MSL drivers.

Return Value

EAX is zero if the hardware options were successfully registered for the adapter. A non-zero value indicates the routine failed to register the hardware options due to a conflict with existing hardware or a bad command line parameter.

Requirements

This routine must only be called at process time (typically during the driver's initialization routine). Interrupts can be in any state on entry and that state will not change during the execution of this routine.

Description

The driver calls *RegisterHardwareOptions* to reserve the hardware configuration options required for a particular physical board (such as port base and range, memory decode addresses and lengths, interrupt number, and DMA channel usage). This routine requires a pointer to an *IOConfigurationStructure* containing the specified hardware options to reserve. If any of the hardware options are already in use, the routine returns an error code.

The reserved hardware options must be released before the driver is removed using the *DeRegisterHardwareOptions* routine.

This routine is normally used in conjunction with the *ParseDriver-Parameters* routine (which parses the configuration options from the load command line).

See Also

DeRegisterHardwareOptions
DriverInitialize
ParseDriverParameters
IOConfigurationStructure
Appendix C, "Obtaining Configuration Information"

Example

```
DriverInitialize   proc

   CPush
   mov     ebp, esp
   pushfd
   cli
   .
   .
   .

;*** Parse Hardware Options ***

   push    [ebp+Parm1]                           ;Screen Handle
   push    [ebp+Parm2]                           ;Command Line pointer
   push    NeedsIOPort0Bit + NeedsInterrupt0Bit ;need I/O port & interrupt
   push    0                                     ;unused
   push    0                                     ;unused
   push    OFFSET AdapterOptions                 ;card options template
   push    0                                     ;unused
   push    OFFSET DriverConfiguration            ;IOConfig structure
   call    ParseDriverParameters                 ;fill out IOConfig Struct
   add     esp, 8 * 4                            ;clean up stack
   or      eax, eax                              ;check status
   jnz     ErrorParsingCommandLine               ;jump on error


;*** Register Hardware Options ***

   push    0                                     ;unused
   push    OFFSET DriverConfiguration            ;IOConfig structure
   call    RegisterHardwareOptions               ;register hardware
   add     esp, 2*4                              ;restore the stack
   or      eax, eax                              ;check status
   jnz     ErrorRegisteringHardware              ;jump on error
```

RegisterServerCommDriver

[Blocking]

Syntax

```
long   RegisterServerCommDriver  (
            struct ResourceTagStructure  *ResourceTag ,
            struct IOConfigurationStructure  *IOConfig ,
            long  (*SendProcedure) ( ) ,
            long  (*BuildSendProcedure) ( ) ,
            long  (*EmergencySendProcedure) ( ) ,
            long  (*ControlProcedure) ( ) ) ;
```

Parameters

ResourceTag
Resource tag with an *MSLSignature* obtained by the driver to register with the Mirrored Server Link interface.

IOConfig
Pointer to the driver's completed IOConfiguration structure.

SendProcedure
Pointer to the *DriverSend* procedure.

BuildSendProcedure
Pointer to the *DriverBuildSend* procedure.

EmergencySendProcedure
Pointer to the *DriverEmergencySend* procedure.

ControlProcedure
Pointer to the *DriverControl* procedure.

Return Value

EAX is zero if the driver is successfully registered. A non-zero indicates failure.

Requirements

This routine may only be called from the Blocking Process level (during the *DriverInitialize* procedure).

Description

This routine is called to register an MSL driver with the Mirrored Server Link interface.

Note: Upon successful completion of the this call, the driver must initialize the global variables, *MaximumCommDriverDataLength* and *PacketSizeDriverCanNowHandle*. Refer to Chapter 3, "Data Structures, Tables, and Variables", for the descriptions of these variables.

Example

```
DriverInitialize   proc

   .
   .
   .

   push  OFFSET DriverControl                  ;DriverControl routine
   push  OFFSET DriverEmergencySend            ;DriverEmergencySend routine
   push  OFFSET DriverBuildSend                ;DriverBuildSend routine
   push  OFFSET DriverSend                     ;DriverSend routine
   push  OFFSET DriverConfiguration            ;DriverIOConfig structure
   push  MSLResourceTag                        ;Resource Tag

   call  RegisterServerCommDriver

   add   esp, 6 * 4                            ;clean up stack
   or    eax, eax                              ;check status
   jnz   ErrorRegisteringDriver                ;jump on error


   mov   MaximumCommDriverDataLength, MAX_PACKET_DATA_SIZE
   mov   PacketSizeDriverCanNowHandle, MAX_PACKET_DATA_SIZE
```

See Also *DriverInitialize, DeRegisterServerCommDriver*

RemovePollingProcedure

[Non-Blocking]

Syntax

```
void   RemovePollingProcedure  (
            void  ( *DriverPollProcedure ) ( void ) ) ;
```

Parameters

DriverPollProcedure
Pointer to a polling procedure defined by the driver that was previously registered with th OS using *AddPolling-ProcedureRTag*.

Return Value

None

Requirements

This routine may only be called at process time. Interrupts may be in any state on entry and will remain unchanged.

Description

This procedure is used to remove a driver's poll routine from the server's list of polling procedures. *RemovePollingProcedure* should be called when a polled driver unloads.

Example

```
push  OFFSET MyDriverPollProc    ;Remove us from poll
call  RemovePollingProcedure     ;List
add   esp, 1 * 4
```

See Also

AddPollingProcedureRTag

ReturnSharedMemoryLinearAddress

Syntax

void ReturnSharedMemoryLinearAddress (
long SharedMemoryLinearAddress) ;

Parameters

SharedMemoryLinearAddress
32-bit address returned by the *GetSharedMemoryLinearAddress* procedure.

Return Value

None

Requirements

Description

This procedure is called to return the 32-bit address obtained from *GetSharedMemoryLinearAddress*. Because of paging by the operating system, this resource should be returned to avoid problems.

Example

```
push  SharedRAMSize
push  SharedRAMPhysicalAddress
call  GetSharedMemoryLinearAddress
add   esp, 2*4
mov   SharedRAMLogicalAddress, eax
   .
   .
   .
push  SharedRAMLogicalAddress
call  ReturnSharedMemoryLinearAddress
add   esp, 1*4
```

See Also

GetSharedMemoryLinearAddress

ScheduleInterruptTimeCallBack

[Non-Blocking, Register-Based Routine]

On Entry

EDX points to a *TimerDataStructure* as shown below.

```
TimerDataStructure          struc
   TLink                    dd ?     ;reserved
   TCallBackProcedure       dd ?
   TCallBackEBXParameter    dd ?
   TCallBackWaitTime        dd ?
   TResourceTag             dd ?
   TWorkWakeUpTime          dd ?     ;reserved
   TSignature               dd ?     ;reserved
TimerDataStructure          ends
```

The reserved fields of this structure are used internally by the NetWare OS and should not be modified by the driver. The remaining fields are filled in as follows:

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

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

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

TResourceTag
Resource tag with a *TimerSignature* acquired by the driver for interrupt time call backs.

On Return

Assume all registers are destroyed.

Requirements

This routine may be called at either process or interrupt time. Interrupts must be disabled on entry and will remain disabled throughout this routine.

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; the driver must reschedule each time it wants another callback. The four fields of the structure that are set by the driver are not changed by the operating system. If the driver reschedules another callback, it does not need to reinitialize these fields.

The MSL driver will typically schedule an interrupt time callback to a driver procedure in order to attempt to redeliver a heldoff message to the OS or to monitor for and recover from timeout conditions.

Example

```
RTagMessage_Timer db 'Timer Callback', 0
IntHoldOffEvent   TimerDataStructure  <,DriverIntHoldOff,,1,,,>



push  TimerSignature
push  OFFSET RTagMessage_Timer
push  ModuleHandle
call  AllocateResourceTag
add   esp, 3 * 4
or    eax, eax
jz    ErrorAllocatingTimerRTag
mov   IntHoldOffEvent.TResourceTag, eax
 .
 .
 .
cli
mov   edx, OFFSET IntHoldOffEvent
call  ScheduleInterruptTimeCallBack
```

See Also

AllocateResourceTag
CancelInterruptTimeCallBack
DriverIntHoldOff
ScheduleNoSleepAESProcessEvent, CancelNoSleepAESProcessEvent
ScheduleSleepAESProcessEvent, CancelSleepAESProcessEvent

ScheduleNoSleepAESProcessEvent

[Non-Blocking]

Syntax

void ScheduleNoSleepAESProcessEvent (
*struct AESEventStructure *EventNode) ;*

Parameters

EventNode - Pointer to an AESEventStructure defined as follows:

```
AESEventStructure        struc
   AESLink               dd   ?
   AESWakeUpDelayAmount  dd   ?
   AESWakeUpTime         dd   ?
   AESProcessToCall      dd   ?
   AESRTag               dd   ?
   AESOldLink            dd   ?
   MessageTimeOutTime    dd   ?
   AdapterTimeOutTime    dd   ?
AESEventStructure        ends
```

AESLink
Used internally by the NetWare OS; do *not* modify this field.

AESWakeUpDelayAmount
Indicates the time interval for waking up your *Timeout* routine in system clock ticks (1 tick is approximately 1/18 sec). Generally, this interval should be small enough to provide reasonable recovery time, but not so small as to affect overall server performance.

AESWakeUpTime
Used internally by the NetWare OS; do *not* modify this field.

AESProcessToCall
A pointer to the routine that will be called once for each *ScheduleNoSleepAESProcessEvent* call.

AESRTag
Resource tag with an *AESProcessSignature* obtained by the MSL driver during initialization.

AESOldLink
Maintained for backward compatibility.

MessageTimeOutTime
Set this field to the value of *ServerCommACKTimeOut* when beginning a message timeout sequence. This value is the maximum time (in ticks) you should wait for the other server's acknowledgment before calling *ServerCommDriverError*.

AdapterTimeOutTime (optional)
This field is used by adapters that support the transmit complete feature. When the driver initiates a transmission, it should set this value to the maximum time (in ticks) to wait for that transmission to complete. This can be used to detect a "dead" adapter.

Return Value

None

Requirements

This routine may be called at either process or interrupt time. Interrupts may be in any state on entry and that state will not be changed during this routine.

Description

ScheduleNoSleepAESProcessEvent sets up a background AES process (Asynchronous Event Scheduler) that will be executed at a desired interval. The specified callback procedure will be called at process time and must be *non-blocking*. (If the callback procedure must call any blocking support procedures, use the *ScheduleSleepAESProcessEvent*.)

The specified procedure will only be called once; the driver must reschedule each time it wants another callback. The fields of the structure that are filled in by the driver are not changed by the operating system. If the driver reschedules another callback, it does not need to reinitialize these fields.

Example

```
AES_RTagMessage   db 'AES Callback', 0
TimeOutEvent      AESEventStructure  <,5,,DriverTimeOut>



push  AESProcessSignature
push  OFFSET AES_RTagMessage
push  ModuleHandle
call  AllocateResourceTag
add   esp, 3 * 4
or    eax, eax
jz    ErrorAllocatingAESRTag
mov   TimeOutEvent.AESRTag, eax
 .
 .
 .
push  OFFSET TimeOutEvent
call  ScheduleNoSleepAESProcessEvent
add   esp, 1 * 4
```

See Also

AllocateResourceTag
CancelNoSleepAESProcessEvent
DriverTimeOut
ScheduleSleepAESProcessEvent, CancelSleepAESProcessEvent
ScheduleInterruptTimeCallBack, CancelInterruptTimeCallBack

ScheduleSleepAESProcessEvent

[Non-Blocking]

Syntax

void ScheduleSleepAESProcessEvent (
*struct AESEventStructure *EventNode) ;*

Parameters

EventNode - Pointer to an AESEventStructure defined as follows:

```
AESEventStructure        struc
   AESLink               dd   ?
   AESWakeUpDelayAmount  dd   ?
   AESWakeUpTime         dd   ?
   AESProcessToCall      dd   ?
   AESRTag               dd   ?
   AESOldLink            dd   ?
   MessageTimeOutTime    dd   ?
   AdapterTimeOutTime    dd   ?
AESEventStructure        ends
```

AESLink
Used internally by the NetWare OS; do *not* modify this field.

AESWakeUpDelayAmount
Indicates the time interval for waking up your callback routine in system clock ticks (1 tick is approximately 1/18 sec). Generally, this interval should be small enough to provide reasonable recovery time, but not so small as to affect overall server performance.

AESWakeUpTime
Used internally by the NetWare OS; do *not* modify this field.

AESProcessToCall
A pointer to the routine that will be called once for each *ScheduleSleepAESProcessEvent* call.

AESRTag
Resource tag with an *AESProcessSignature* obtained by the MSL driver during initialization.

AESOldLink
Maintained for backward compatibility.

MessageTimeOutTime
Set this field to the value of *ServerCommACKTimeOut* when beginning a message timeout sequence. This value indicates the maximum time (in ticks) you should wait for the other server's acknowledgment before calling *ServerCommDriverError*.

AdapterTimeOutTime (optional)
This field is used by adapters that support the transmit complete feature. When the driver initiates a transmission, it should set this value to the maximum time (in ticks) to wait for that transmission to complete. This can be used to detect a "dead" adapter.

Return Value

None

Requirements

This routine may be called at either process or interrupt time. Interrupts may be in any state on entry and that state will not be changed during this routine.

Description

ScheduleSleepAESProcessEvent sets up a background AES process (Asynchronous Event Scheduler) that will be executed at a designated interval. The specified callback procedure will be called at process time and may perform blocking calls during its execution. (If the callback procedure does not use any blocking support procedures, use the *ScheduleNoSleepAESProcessEvent* procedure instead.)

The specified procedure will only be called once; the driver must reschedule each time it wants another callback. The fields of the structure that are filled in by the driver are not changed by the operating system. If the driver reschedules another callback, it does not need to reset these fields.

Example

```
AES_RTagMessage   db  'AES Callback', 0
HoldOffEvent      AESEventStructure <,0,,DriverHoldOff>



push  AESProcessSignature
push  OFFSET AES_RTagMessage
push  ModuleHandle
call  AllocateResourceTag
add   esp, 3 * 4
or    eax, eax
jz    ErrorAllocatingAESRTag
mov   HoldOffEvent.AESRTag, eax
 .
 .
 .
push  OFFSET HoldOffEvent
call  ScheduleSleepAESProcessEvent
add   esp, 1 * 4
```

See Also

AllocateResourceTag
CancelSleepAESProcessEvent
DriverHoldOff
ScheduleNoSleepAESProcessEvent, CancelNoSleepAESProcessEvent
ScheduleInterruptTimeCallBack, CancelInterruptTimeCallBack

SendServerCommCompletedPointer

[Non-Blocking, Register-Based Routine]

Syntax *call [SendServerCommCompletedPointer]*

On Entry

EBP is the number of messages being Acknowledged.

On Return

Assume all registers are destroyed.

Requirements This routine is called from the Interrupt level. Interrupts must be disabled on entry and will remain disabled.

Description *SendServerCommCompletedPointer* is a global variable containing a pointer to the current routine used to notify the OS of the number of messages being acknowledged by the other server. The SFT III operating system requires all messages to be delivered and acknowledged in the order they were given to the driver.

After the driver notifies the OS of the acknowledgements, it must transmit any messages that the OS queued up while the driver was busy transmitting that last message packet. (Refer to the *GetNext-PacketPointer* procedure for a description of this process.)

Example

```
DriverISR   proc
    .
    .
    .
;*************************************************************************
;* Acknowledgement Received                                              *
;*************************************************************************

ISRAckReceived:

    cmp     MessageInProgress, TRUE                 ;validate ack
    jne     CheckAdapterStatus                      ;

    ;*********************************************************************
    ;* Cancel Message TimeOut Sequence                                   *
    ;*********************************************************************

    mov     MessageInProgress, FALSE                ;clear flag
    mov     TimeOutEvent.MessageTimeOutTime, 0      ;stop message timer

    ;*********************************************************************
    ;* Notify OS of the acknowledgement(s)                               *
    ;*********************************************************************

    mov     ebp, TxPacketMessageCount               ;get # of messages sent
    add     ReceiveAckCount, ebp                    ;update statistics counter
    call    [SendServerCommCompletedPointer]        ;notify OS of ACKs
    .                                               ;(use indirect call)
    .
    .
```

See Also *DriverISR*
GetNextPacketPointer

ServerCommDriverError

[Non-Blocking]

Syntax

void ServerCommDriverError (ErrorCode) ;

Parameters

ErrorCode
Cause for the error in mirrored server communications.

```
HARDWARE_ERROR            equ   00h
TIME_OUT_ERROR            equ   01h
OTHER_SERVER_DEAD_ERROR   equ   02h
```

(These error codes are found in the include file MSL.INC)

Return Value

None

Requirements

This routine may be called at either Interrupt or Process time. Interrupts must be disabled on entry.

Description

The MSL driver calls this procedure to notify the operating system of an error in mirrored server communications. This error may be due to an unrecoverable hardware failure, an unacknowledged message transmission, or an emergency notification from the other server.

If the MSL driver detects an unrecoverable hardware error, call *ServerCommDriverError* with an error code of 0 (`HARDWARE_ERROR`).

If the MSL driver sends a message packet, and does not receive a message acknowledgement from the other server before the timeout limit is reached, *ServerCommDriverError* should be called with an error code of 1 (`TIME_OUT_ERROR`).

If the MSL driver receives an emergency notification from the other mirrored server, *ServerCommDriverError* should be called with an error code of 2 (`OTHER_SERVER_DEAD_ERROR`).

Important: Normally, the driver calls *ServerCommDriverError* immediately upon detection of an error. However, when in a message holdoff state (see *ReceiveServerCommPointer*), any heldoff messages *that have already been acknowledged* must be delivered to the OS before reporting the error. The driver should flag the error, finish delivering all acknowledged messages, and only then notify the OS of the detected error. The driver must report errors in this manner to preserve the mirrored state of the servers.

Note: It is not necessary to call *ServerCommDriverError* in the *DriverRemove* routine. The OS is notified that the MSL driver is being unloaded when *DeRegisterServerCommDriver* is called.

Example

```
push  TIME_OUT_ERROR                    ;error code
call  ServerCommDriverError             ;report error to OS
add   esp, 1*4                          ;clean up stack
```

SetHardwareInterrupt

[Non-Blocking]

Syntax

```
long  SetHardwareInterrupt (
          long   HardwareInterruptLevel ,
          void  ( *InterruptProcedure ) ( void ) ,
          struct ResourceTagStructure  *ResourceTag ,
          long   EndOfChainFlag ,
          long   ShareFlag ,
          long  *EOIFlag ) ;
```

Parameters

HardwareInterruptLevel
The hardware interrupt level.

InterruptProcedure
Pointer to the interrupt procedure that will be assigned to the specified interrupt.

ResourceTag
Resource tag with an *InterruptSignature* acquired by the driver for setting up interrupt service. (see *AllocateResourceTag*)

EndOfChainFlag
This flag indicates whether a shared interrupt ISR is placed on the front or the back of the chained interrupt queue. If this flag is set to 0, the ISR is to be placed on the front of the queue (non-shared interrupts should use 0). If this flag is set to 1, and the *ShareFlag* is also set to 1, the ISR should be placed at the end of the queue.

```
CHAIN_FIRST 0
CHAIN_LAST  1
```

ShareFlag
Flag indicating whether interrupts may be shared by the device and the driver with other boards and drivers. A value of 1 indicates the interrupt can be shared; a value of 0 indicates the interrupt is non-sharable.

```
CHAIN_SHARE_BIT      1
CHAIN_SET_REAL_MODE  4 (also set for real mode)
```

EOIFlag
Pointer to a double-word flag that, on return from this procedure, indicates if a second EOI is required for this interrupt.

If on return, this flag is zero, only one EOI will be required for the interrupt. If this flag is non-zero, 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.

Return Value

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

Requirements

This procedure must only be called at process time. Interrupts must be disabled on entry and will remain disabled throughout this routine.

Description

SetHardwareInterrupt allocates the specified interrupt and provides the OS interrupt handler with the driver's ISR entry point.

The operating system fields the actual interrupt. When the driver's ISR is called, the direction is cleared, system interrupts are disabled, all registers are saved, and segment registers are set up. The driver only needs to EOI the PIC, service the interrupt, and return (do not use `iretd`, since the OS issues an `iretd` upon completion).

Note: If the driver needs to change the direction flag, it should do so with interrupts disabled and then restore the direction flag to the cleared state.

Example

```
DriverInitialize   proc
    .
    .
    .
   push  OFFSET ExtraEOIFlag
   push  CHAIN_SET_REAL_MODE
   push  0
   push  InterruptResourceTag
   push  OFFSET DriverISR
   movzx eax, BYTE PTR DriverConfiguration.CInterrupt
   push  eax

   call  SetHardwareInterrupt

   add   esp, 6 * 4
   or    eax, eax
   jnz   ErrorSettingInterrupt
    .
    .
    .
DriverInitialize   endp


DriverISR   proc
    .
    .
    .
   ret

DriverISR   endp
```

See Also

ClearHardwareInterrupt, AllocateResourceTag
DriverInitialize

UnRegisterEventNotification

Syntax *long UnRegisterEventNotification (long EventID) ;*

Parameters

EventID
Value obtained when *RegisterForEventNotification* was called.

Return Value

EAX is zero (0) if successful; a value of one (1) indicates failure.

Requirements This routine should be called when the driver is being unloaded (during the *DriverRemove* procedure).

Description *UnRegisterEventNotification* is called to unhook the driver from event notification.

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

Example

```
DriverRemove   proc
   .
   .
   .
   push  EventID                          ;Unhook from OS exit
   call  UnRegisterEventNotification      ;Call OS to unhook
   add   esp, 1*4                         ;Clear stack
   .
   .
   .
DriverRemove   endp
```