7

MSM Procedures and Macros

Introduction

This chapter describes the MSM procedures and macros provided as tools for HSM developers. These MSM procedures, along with the topology specific procedures described in Chapter 6, manage the primary details of interfacing the HSM to the Link Support Layer. The procedures and macros in this chapter are media independent and handle generic initialization and run-time issues. The macros included in this section are defined in the MSM.INC file.

MSMAlertFatal

On Entry

EBP	Pointer to Adapter Data Space
ECX	Possible argument #1
EDX	Possible argument #2
ESI	Pointer to null terminated error message
Interrupts	can be in any state, but will be disabled during the call
Call	at process or interrupt time

On Return

Interrupts	are in the same state as when the routine was called
Note	EBX and EBP are preserved

Description

The HSM can call *MSMAlertFatal* during regular operation (run-time) to notify the operating system of driver hardware or software problems. An error severity level of "fatal" will be reported with the developer-provided error message. This routine will not relinquish control to other procedures during execution.

The "Possible Arguments #1 and #2" above are used here the same way in which they are used in the printf routine in C. If there are no format specifications in the string, ECX and EDX are ignored.

This routine has added functionality which supports an additional string format. If the string is preceded by a word size error number in the range of 100-999, the MSM will print the driver name, the platform name (NW for NetWare 386), the decimal error number, and the instance of the board, before printing the specified string. (See Appendix H for a listing of standard messages.)

Example

```
ErrorMessage   dw   105
               db  "Board did not respond to multicast update.", 0
   •
   •
   •
lea   ESI, ErrorMessage
call  MSMAlertFatal
```

The example above would output the following message if the adapter is an NE2000 and was the first NE2000 registered:

```
NE2000-NW-105-Adapter 1:  Board did not respond to multicast update.
```

MSMAlertWarning

On Entry

EBP	Pointer to the Adapter Data Space
ECX	Possible argument #1
EDX	Possible argument #2
ESI	Pointer to a null terminated error message
Interrupts	can be in any state, but will be disabled during the call
Call	at process time or interrupt time

On Return

Interrupts	are in the same state as when the routine was called
Note	EBX and EBP are preserved

Description

The HSM can call *MSMAlertWarning* during regular operation (run-time) to notify the operating system of driver hardware or software problems. An error severity level of "warning" will be reported with the developer-provided error message. This routine will not relinquish control to other procedures during execution.

The "Possible Arguments #1 and #2" above are used here the same way in which they are used in the printf routine in "C." If there are no format specifications in the string, ECX and EDX are ignored.

This routine has added functionality which supports an additional string format. If the string is preceded by a word size error number in the range of 100-999, the MSM will print the driver name, the platform name (NW for NetWare 386), the decimal error number, and the instance of the board, before printing the specified string. (See Appendix H for a listing of standard messages.)

Example

```
ErrorMessage   dw    105
               db   "Board did not respond to multicast update.",0
•
•
•
lea   ESI, ErrorMessage
call  MSMAlertWarning
```

The example above would output the following message if the adapter is an NE2000 adapter and was the first NE2000 registered:

```
NE2000-NW-105-Adapter 1:  Board did not respond to multicast update.
```

MSMAlloc

On Entry

EBP	Pointer to the Adapter Data Space
EAX	Number of bytes of memory to allocate
Interrupts	can be in any state (but might be enabled during the call)
Call	at process time only

On Return

EAX	Pointer to the allocated buffer. (zero = failure)
Interrupts	are in the same state as when the routine was called
Note	EBX, EBP, ESI, and EDI are preserved

Description

The HSM may use this call to allocate memory at process time. *MSMAlloc* returns a pointer to the allocated buffer in EAX. If the routine was unsuccessful, EAX will be zero. It is the responsibility of the HSM to return this buffer at shutdown using *MSMFree*.

If the *DriverParameterBlock* variable, *DriverNeedsBelow16Meg*, was initialized to any non-zero value (see chapter 3), the MSM will allocate memory below the 16 megabyte boundary.

Example

```
mov   eax, UserBufferSize
call  MSMAlloc
or    eax,eax
jz    ErrorAllocatingBuffer
```

MSMAllocPages

On Entry

EAX	Number of bytes of memory to allocate
Interrupts	can be in any state
Call	at process time only

On Return

EAX	Pointer to the allocated buffer. (zero = failure)
Interrupts	are in the same state as when the routine was called
Note	EBX, EBP, ESI, and EDI are preserved

Description

The HSM may use this call to allocate a memory buffer on a 4K page boundary at process time. *MSMAllocPages* returns a pointer to the allocated buffer in EAX. If the routine was unsuccessful, EAX will be zero. It is the responsibility of the HSM to return this buffer at shutdown using *MSMFreePages*.

If the *DriverParameterBlock* variable, *DriverNeedsBelow16Meg*, was initialized to any non-zero value (see chapter 3), the MSM will allocate memory below the 16 megabyte boundary.

Example

```
mov   eax, UserPageBufferSize
call  MSMAllocPages
or    eax,eax
jz    ErrorAllocatingBuffer
```

MSMAllocateRCB

On Entry

EBP	Pointer to the Adapter Data Space
ESI	Packet Size including all the headers if known; otherwise use the maximum packet size.
Interrupts	can be in any state
Execute	at process or interrupt time

On Return

ESI	Pointer to an RCB (non-fragmented)
Flags	zero flag is set if routine is successful
Interrupts	are disabled
Note	EAX is destroyed; all other registers are preserved

Description

The HSM uses *MSMAllocateRCB* to allocate an RCB for a packet it has received or to preallocate an RCB for a packet it will be receiving. The RCB returned will be non-fragmented (see Chapter 4) and will be large enough to hold the received packet frame. The length passed in register ESI should also include the length of all protocol and hardware headers. If an RCB is not available, the MSM will increment the *NoECBAvailableCount* statistics counter and the packet should be discarded.

HSMs that support bus-mastering DMA adapters should use this routine to preallocate RCBs. In this case, the HSM should set ESI to the maximum packet size specified by the *MLIDMaximumSize* field of the configuration table before using *MSMAllocateRCB*.

After the adapter has copied the packet into the *RCBDataBuffer* field of the RCB, the HSM should use either *<TSM>ProcessGetRCB* or *<TSM>FastProcessGetRCB* to return the RCB to the MSM. If the adapter is ECB aware and has previously filled in all the RCB fields according to the ODI specification, the HSM should call *<TSM>RcvComplete* or *<TSM>FastRcvComplete*.

Note: If the *DriverParameterBlock* variable, *DriverNeedsBelow16Meg*, was initialized to any non-zero value (see chapter 3), the MSM will allocate the RCB in memory below the 16 megabyte boundary.

Special Instructions

Ethernet
The HSM should start copying the packet from the 6 byte destination field of the media header into the *RCBDataBuffer* field of the RCB.

Token-Ring
The HSM should start copying the packet from the Access Control byte of the media header into the *RCBDataBuffer* field of the RCB.

FDDI
The HSM should start copying the packet from the Frame Control byte of the media header into the *RCBDataBuffer* field of the RCB.

PCN2L
This routine is not used for PCN2 drivers.

Example

```
; ebx = ptr to Frame Data Space

mov   esi,[ebx].MLIDMaximumSize     ; ESI = Max Packet size
call  MSMAllocateRCB                ; Get an RCB
jnz   UnableToAllocateRCB           ; Jump if unsuccessful
```

MSMDisableHardwareInterrupt (macro)

On Entry

EBP	Pointer to the Adapter Data Space
Interrupts	are disabled
Execute	at process or interrupt time

On Return

Interrupts	are unchanged
Note	EAX, ECX and EDX are destroyed

Description

This macro disables the adapter's interrupt line on the Programmable Interrupt Controller (PIC). This macro should not be used when sharing the interrupt.

Important ! Only use this macro if interrupts can not be enabled and disabled at the adapter hardware level. If you can control interrupts at the adapter, you should implement the *DriverEnableInterrupt* and *DriverDisable-Interrupt* routines described in Chapter 5. Drivers that control interrupts at the adapter can be transported more easily to other OS platforms where access to the PIC is restricted.

Example

```
DriverISR     proc
   MSMDisableHardwareInterrupt
   MSMDoEndOfInterrupt
   inc    [ebp].InDriverISR             ; Set for DriverSend
    .
    .    (Service the adapter)
    .
   dec    [ebp].InDriverISR             ; Clear InISR flag
   MSMEnableHardwareInterrupt
   MSMServiceEventsAndRet               ; Let LSL unqueue returned
DriverISR     endp

DriverSend    proc
   cmp    [ebp].InDriverISR, 0          ; Called from DriverISR?
   jnz   DriverStartSend                ; Jump if so
   MSMDisableHardwareInterrupt

DriverStartSend:
    .
    .    (Send the packet)
    .
   cmp    [ebp].InDriverISR, 0          ; Called from DriverISR?
   jnz    <TSM>SendComplete             ; We're finished if so.

   MSMEnableHardwareInterrupt
   jmp    <TSM>FastSendComplete         ; Give back TCB and service events
DriverSend    endp
```

MSMDriverRemove

On Entry

EAX	*DriverModuleHandle* from the *DriverParameterBlock* structure
Interrupts	can be in any state
Call	at process time only

On Return

EAX	is preserved
Interrupts	are unchanged

Description

This routine is called by the HSM's *DriverRemove* procedure to de-register the driver and return all driver resources. *MSMDriverRemove* will call the HSM's *DriverShutdown* routine before returning.

Example

```
DriverRemove   proc

   Cpush                        ; Macro to save "C" registers
   mov   eax, DriverModuleHandle ; Get Module Handle from Parameter Block
   call  MSMDriverRemove         ; De-register the driver
   Cpop                         ; Restore "C" registers

DriverRemove   endp
```

MSMDoEndOfInterrupt (macro)

On Entry

EBP	Pointer to the Adapter Data Space
Interrupts	are disabled
Execute	at interrupt time

On Return

Interrupts	are unchanged
Note	EAX and ECX are destroyed

Description

This macro is used in *DriverISR* to send an EOI to the Programmable Interrupt Controller (PIC). *MSMDoEndofInterrupt* calls the operating system to service the primary and secondary PICs.

Note: Novell recommends that the developer use this macro rather than programming the PIC directly. This will allow the HSM to run on a wider variety of PCs.

Example

(see example for the macro *MSMDisableHardwareInterrupt*)

MSMEnableHardwareInterrupt (macro)

On Entry

EBP	Pointer to the Adapter Data Space
Interrupts	are disabled
Execute	at process or interrupt time

On Return

Interrupts	are unchanged
Note	EAX, ECX and EDX are destroyed

Description This macro enables the adapter's interrupt line on the Programmable Interrupt Controller (PIC).

Important ! Only use this macro if interrupts can not be enabled and disabled at the adapter hardware level. If you can control interrupts at the adapter, you should implement the *DriverEnableInterrupt* and *DriverDisable-Interrupt* routines described in Chapter 5. Drivers that control interrupts at the adapter can be transported more easily to other OS platforms where access to the PIC is restricted.

Example

(see example for the macro *MSMDisableHardwareInterrupt*)

MSMEnablePolling

On Entry

EBP	Pointer to the Adapter Data Space
Interrupts	can be in any state
Call	at process or interrupt time (usually called during initialization)

On Return

EAX	Zero if successful; otherwise EAX points to an error message that the driver must print using *MSMPrintString* before returning to the operating system with EAX non-zero.
Zero Flag	Set if successful; otherwise an error occurred.
Interrupts	are unchanged
Note	EBX and EBP are preserved

Description

If the HSM's board service routine is poll-driven, this routine can be used during *DriverInit* to enable the operating system to periodically call *DriverPoll*. The *DriverPoll* routine polls the adapter to determine if any send or receive events have occurred.

This routine will not relinquish control to other procedures during execution.

Example

```
DriverInit  proc
      ⋮
   call  MSMEnablePolling          ; Enable DriverPoll
   jnz   EnablePollingError
      ⋮
DriverInit  endp
```

MSMEndCriticalSection (macro)

On Entry

EBP	Pointer to the Adapter Data Space
Interrupts	can be in any state
Execute	at process or interrupt time

On Return

Interrupts	are unchanged
Note	all registers are preserved

Description

The *MSMStartCriticalSection* and *MSMEndCriticalSection* macros are used to prevent the TSM from calling *DriverSend* while it is performing critical operations. This allows interrupts to be enabled in the *DriverSend* and/or *DriverISR* routine.

When a TCB needs to be sent, the TSM usually calls *DriverSend*. However, *DriverSend* may be reading bytes from the card and starting a send at this point could corrupt data. If the HSM is in a critical section, the TSM queues the packet instead of calling *DriverSend*.

Note: The example on the following page illustrates the use of the macros, *MSMStartCriticalSection* and *MSMEndCriticalSection*. However, Novell recommends that interrupts remain disabled during the *DriverSend* and *DriverISR* routines.

Important ! If you can control interrupts at the adapter, you should implement the *DriverEnableInterrupt* and *DriverDisableInterrupt* routines described in Chapter 5. Drivers that control interrupts at the adapter should not use *MSMEnableHardwareInterrupt*, *MSMDisableHardwareInterrupt*, or *MSMDoEndOfInterrupt* macros.

Example

```
DriverISR     proc

   MSMDisableHardwareInterrupt
   MSMDoEndOfInterrupt
   inc   [ebp].InDriverISR            ; Set for DriverSend
   MSMStartCriticalSection            ; Inform MSM before
   sti                                ;  enabling interrupts
    .
    .    (Service the adapter)
    .
   cli
   dec   [ebp].InDriverISR            ; Clear InISR flag
   MSMEndCriticalSection              ; Exiting Critical Section
   MSMEnableHardwareInterrupt
   MSMServiceEventsAndRet             ; Let LSL unqueue returned

DriverISR     endp



DriverSend    proc

   cmp   [ebp].InDriverISR, 0         ; Called from DriverISR?
   jnz   DriverStartSend              ; Jump if so
   MSMDisableHardwareInterrupt
   MSMStartCriticalSection            ; Inform MSM before enabling
   sti                                ;  interrupts
DriverStartSend:
    .
    .    (Send the packet)
    .
   cmp   [ebp].InDriverISR, 0         ; Called from DriverISR?
   jnz   <TSM>SendComplete            ; We're finished if so.
   cli                                ; Disable interrupts before
   MSMEndCriticalSection              ;  exiting critical section
   MSMEnableHardwareInterrupt
   jmp   <TSM>FastSendComplete        ; Give back TCB and service events

DriverSend    endp
```

MSMFree

On Entry

EBP	Pointer to the Adapter Data Space
EAX	Pointer to the buffer
Interrupts	can be in any state
Call	at process or interrupt time

On Return

Interrupts	are unchanged
Note	EBX, EBP, ESI, and EDI are preserved

Description

The HSM must use this routine to return any memory allocated using *MSMAlloc* before the driver is permanently shutdown. If the driver is being permanently shutdown, the HSM's *DriverShutdown* routine would have been called with ECX equal to zero.

Example

```
DriverShutdown proc
   ⋮
   or    ecx,ecx
   jnz   PartialShutdown
   mov   eax,UserBuffer
   call  MSMFree
   ⋮
DriverShutdown endp
```

MSMFreePages

On Entry

EAX	Pointer to the buffer
Interrupts	can be in any state
Call	at process time only

On Return

Interrupts	are unchanged
Note	EBX, EBP, ESI, and EDI are preserved

Description

The HSM must use this routine to return any memory buffers allocated on 4K page boundaries using *MSMAllocPages* before the driver is permanently shutdown. If the driver is being permanently shutdown, the HSM's *DriverShutdown* routine would have been called with ECX equal to zero.

Example

```
DriverShutdown proc
      ⋮
   or    ecx,ecx
   jnz   PartialShutdown
   mov   eax,UserPageBuffer
   call  MSMFreePages
      ⋮
DriverShutdown endp
```

MSMGetCriticalStatus (macro)

On Entry

EBP	Pointer to the Adapter Data Space
Interrupts	can be in any state
Execute	at process or interrupt time

On Return

EAX	Non-zero if critical section is in progress
Interrupts	are unchanged
Note	all other registers are preserved

Description

MSMGetCriticalStatus returns a value indicating the critical section status of the HSM. If this value is zero, the HSM is not in a critical section. A non-zero value indicates that a critical section is in progress. See *MSMStartCriticalSection* and *MSMEndCriticalSection*.

Example

```
DriverINTCallBack    proc

   MSMGetCriticalStatus    ; EAX = Critical Status
   or    eax,eax           ; In a critical section?
   jnz   ExitCallBack      ; Jump if so
    .
    .    (Process time out)
    .

ExitCallBack:
   ret

DriverINTCallBack    endp
```

MSMGetCurrentTime (macro)

On Entry

Interrupts	can be in any state
Execute	at process or interrupt time

On Return

EAX	current tick count
Interrupts	are unchanged
Note	all other registers are preserved

Description

MSMGetCurrentTime determines the elapsed time (using the current relative time) for some of the HSM-related activities (for example, *TimeOutCheck*). The value returned at the start of an operation subtracted from the current time 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

```
mov   edx, [ebp].Command         ; Let board attempt to
mov   al, Board_Transmit         ; transmit packet again
out   dx, al
MSMGetCurrentTime                ; EAX = current time.
mov   [ebp].TxStartTime, eax     ; Store new timeout
```

MSMGetHardwareBusType (macro)

On Entry

Interrupts	can be in any state
Execute	at process or interrupt time

On Return

EAX	Bus Type (see Completion Codes below)
Interrupts	are unchanged
Note	all other registers are preserved

Completion Codes

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

Description

MSMGetHardwareBusType returns a value that indicates the server's bus type. This macro allows an HSM to be written so that it can be used for boards with different bus types.

Note: The bit positions of the completion code do not correspond to those used in the *MLIDFlags* field of the configuration table.

Example

```
MSMGetHardwareBusType          ; EAX contains the bus type
cmp   eax, 0                   ; ISA bus?
jz    DoNotScanForSlots        ; Jump if it is
```

MSMGetProcessorSpeedRating (macro)

On Entry

Interrupts	can be in any state
Execute	at process or interrupt time

On Return

EAX	contains a value representing the relative processor speed of the machine
Interrupts	are unchanged
Note	all other registers are preserved

Description

MSMGetProcessorSpeedRating determines the relative processor speed; the larger the value returned, the faster the processor is operating.

Note: Although this procedure provides a means for calculating timing loop delays, this routine should never be used unless it is impossible to enable interrupts and use *GetCurrentTime*. Novell recommends that timing loops be avoided whenever possible.

Example

```
MSMGetProcessorSpeedRating ; EAX = Processor Speed
xor   edx, edx             ; Clear high dword of dividend
mov   ecx, 100             ; Divisor = 100
idiv  ecx                  ; EAX = Speed / 100
mov   ecx, 30000h          ; EAX = (Speed/100) * 30000h
imul  eax, ecx
mov   LoopCounter, eax     ; Save it
```

MSMGetRealModeWorkspace (macro)

Macro Parameters

Semaphore	dword	offset
ProtectedModeAddress	dword	offset
RealModeSegment	word	offset
RealModeOffset	word	offset
WorkSpaceSize	dword	offset

On Entry

Interrupts	can be in any state
Execute	at process or interrupt time

On Return

Interrupts	are unchanged
Note	EBX, EBP, ESI and EDI are preserved

Description

The *MSMGetRealModeWorkSpace* macro is used in conjunction with *MSMRealModeInterrupt* to allow the HSM to execute BIOS interrupts. The following example illustrates using *MSMGetRealModeWorkSpace, MSMPSemaphore, MSMRealModeInterrupt* and *MSMVSemaphore* to make a BIOS call in order to access information about EISA slot configurations.

The input and output parameter structures for the example are defined as follows:

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

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

Example

```
;***Real Mode Access Workspace variables***

WSSem          dd   0 ; Real mode semaphore
WSProtAddr     dd   0 ; Protected mode address
WSRealSeg      dw   0 ; Real mode segment
WSRealOff      dw   0 ; Real mode offset
WSSize         dd   0 ; Workspace Size

InputParms     InputStructure  <>
OutputParms    OutputStructure <>


;***Read the configuration from the EISA BIOS***

   MSMGetRealModeWorkSpace  WSSem WSProtAddr WSRealSeg WSRealOff WSSize
   MSMPSemaphore  WSSem              ; Lock the workspace
   movzx ecx, [ebx].MLIDSlot         ; Start with Block 0

ReadConfigBlockLoop:
   push  ecx                         ; Save Block
   lea   esi, InputParms             ; ESI -> Input Registers
   mov   [esi].IAXRegister, 0D801h   ; AH = 0D8h, AL = 01
   mov   [esi].ICXRegister, cx       ; CH = Block, CL = Slot
   movzx eax, WSRealOffset
   mov   [esi].ISIRegister, ax       ; SI = Real Mode Offset
   movzx eax, WSRealSegment
   mov   [esi].IDSRegister, ax       ; DS = Real Mode Segment
   mov   [esi].IntNumber, 15h        ; BIOS Interrupt 15h

   MSMRealModeInterrupt  InputParms OutputParms

   pop   ecx                         ; Restore Block/Slot number
   cmp   eax, 0                      ; Was interrupt successful?
   jnz   RealModeInterruptError      ; Jump if not
   cmp   byte ptr OutputParms.OAXRegister + 1, 81h
   je    ExitReadConfigLoop          ; Jump if last configuration block
   cmp   byte ptr OutputParms.OAXRegister + 1, 0
   jne   RealModeInterruptError      ; Jump if int not successful
   mov   esi, WSProtAddr             ; ESI -> Block
   movzx edx, byte ptr [esi + 0b2h]  ; EDX = possible int
   and   dl, ISOLATE_INT_MASK        ; Mask off interrupt field
   or    edx, edx                    ; Interrupt?
   jz    ExitReadConfigLoop          ; Jump out if not
   movzx eax, byte ptr [esi + 22h]   ; EAX = function info
   test  al, EISA_INT_FUNCTION_BIT   ; Valid int?
   jnz   StoreInterruptLevel         ; Jump if so
   inc   ch                          ; CH = Next Block
   jmp   ReadConfigBlockLoop         ; Try again

StoreInterruptLevel:
   mov   [ebx].MLIDInterrupt, dl     ; Copy int to configuration table

ExitReadConfigLoop:
   MSMVSemaphore    WSSemaphore      ; Clear Semaphore
```

MSMInitAlloc

On Entry

EAX	Number of bytes of memory to allocate
Interrupts	can be in any state
Call	at process time only

On Return

EAX	Pointer to the allocated buffer. (zero = failure)
Interrupts	are in the same state as when the routine was called (but might have been enabled during the call if *DriverNeeds-Below16Meg* is non-zero)
Note	EBX, EBP, ESI, and EDI are preserved

Description

HSMs must use the *MSMInitAlloc* routine if they must allocate memory prior to calling *MSMRegisterHardwareOptions*. If successful, *MSMInitAlloc* returns a pointer to the allocated buffer in EAX. If the routine was unsuccessful, EAX will be zero.

If the driver also frees the allocated buffer prior to calling *MSMRegisterHardwareOptions* it must use the *MSMInitFree* routine. (Use the *MSMFree* routine to release the buffer any time after *MSMRegisterHardwareOptions* is called.)

If the *DriverParameterBlock* variable, *DriverNeedsBelow16Meg*, was initialized to any non-zero value (see chapter 3), the MSM will attempt to allocate memory below the 16 megabyte boundary.

Example

```
DriverInit  proc
   .
   .
   .
   mov   eax, UserBufferSize
   call  MSMInitAlloc
   or    eax, eax
   jz    ErrorAllocatingBuffer
   mov   UserBuffer, eax
   .
   .
   .
   mov   eax, UserBuffer
   call  MSMInitFree
   .
   .
   .

   call     MSMRegisterHardwareOptions
```

MSMInitFree

On Entry

EAX	Pointer to the buffer to free (must have been previously allocated using *MSMInitAlloc*)
Interrupts	can be in any state
Call	at process time only

On Return

Interrupts	are preserved
Note	EBX, EBP, ESI, and EDI are preserved

Description

HSMs must use the *MSMInitAlloc* routine during initialization, if they must allocate memory prior to calling *MSMRegisterHardwareOptions*. If the driver also frees the allocated buffer prior to calling *MSMRegisterHardwareOptions* it must use the *MSMInitFree* routine. (Use *MSMFree* instead of this routine to release the buffer any time after *MSMRegisterHardwareOptions* is called.)

Example

```
DriverInit  proc
   .
   .
   .
   mov   eax, UserBufferSize
   call  MSMInitAlloc
   or    eax, eax
   jz    ErrorAllocatingBuffer
   mov   UserBuffer, eax
   .
   .
   .
   mov   eax, UserBuffer
   call  MSMInitFree
   .
   .
   .

   call     MSMRegisterHardwareOptions
```

MSMParseCustomKeywords

On Entry

ESI	Pointer to the *DriverParameterBlock*

On Return

EBX	is preserved
Zero Flag	is cleared if a "T_REQUIRED" custom keyword was not entered on the command line or by user after being prompted.

Description

Drivers can define keywords that allow custom parameters or flags to be entered from the **"load"** command-line. (Refer to Chapter 3, page 3-37, for a complete description of how to define custom keywords.)

Custom keywords are normally processed during initialization when *DriverInit* calls *MSMParseDriverParameters*. If the driver must have custom keywords processed earlier in initialization, the *DriverInit* routine can call *MSMParseCustomKeywords*.

Note: *MSMParseDriverParameters* will still call custom keyword procedures even if *MSMParseCustomKeywords* called them earlier.

The MSM parses the command-line for custom keywords and calls the procedure corresponding to that keyword. Requirements for custom keyword procedures are described in the next section.

Custom Keyword Procedure

When the MSM calls a custom keyword procedure, the values of the registers on entry will vary depending on which keyword parsing flags (if any) were used. Page 3-39 of Chapter 3 describes the parsing flags and how they are used.

On Entry

EDX is non-zero if a T_REQUIRED keyword was found on the original command-line.

EDX is zero if a T_REQUIRED keyword was not found on the original command-line and the user had to be prompted for information.

T_REQUIRED - The keyword must be entered. If it doesn't exist on the command-line or configuration file, the user will be prompted for it. If the users does not enter a value, *MSMParseCustomKeywords* will return with an error.

T_STRING - The Keyword Routine will be called with a pointer to the beginning of the string that matched the keyword text.

Example: `load <driver> custom int=3`

Routine called with ESI pointing to "custom int=3"

T_NUMBER - The Keyword Routine will be called with the value entered on the command-line in EAX. The user must enter a decimal number.

Example: `load <driver> custom=100`

Routine called with EAX = 64h

T_HEX_NUMBER - The Keyword Routine will be called with the value entered on the command-line in EAX. The user must enter a hexadecimal number.

Example: `load <driver> custom=100`

Routine called with EAX = 100h

T_HEX_STRING - The Keyword Routine will be called with ESI pointing to a six byte value that was entered on the command-line. The user must enter this string using hexadecimal numbers.

Example: `load <driver> custom=01020304`

Routine called with ESI -> 00, 00, 01, 02, 03, 04

The following is an example of a driver for an adapter that may require memory below 16 megabytes depending on information read from a port. The example will prompt the user for an I/O port and determine whether it needs memory below 16 megabytes or not.

Example

```
OSDATA segment rw public 'DATA'



DriverParameterBlock       label  dword
    .
    .
    .
  DriverNumKeywords        dd     1
  DriverKeywordText        dd     KeywordTextTable
  DriverKeywordTextLen     dd     KeywordTextLenTable
  DriverProcessKeywordTab  dd     KeywordProcedureTable
    .
    .
    .
;DriverParameterBlockEnd



KeywordTextTable           dd     PortKeyword

KeywordTextLenTable        dd     PortKeywordLen

KeywordProcedureTable      dd     PortKeywordRoutine



;--------------------------------------------------------------------------------
; Define Keywords and related Parameters
;--------------------------------------------------------------------------------

PortKeyword                db     'PORT'
PortKeywordLen             equ    ($ - PortKeyword) OR T_HEX_NUMBER OR T_REQUIRED
                           dd     300                 ; Min port value
                           dd     360                 ; Max port value
                           dd     PortDefault         ; Default Port
                           dd     PortValid           ; Valid characters
                           dd     PortPrompt          ; Prompt string

PortDefault                db     "300", 0
PortValid                  db     "0..9A..F", 0       ; Hex digits only
Port                       db     "Enter the Port Number: ", 0


;--------------------------------------------------------------------------------
; Define some variables used by custom keyword routine
;--------------------------------------------------------------------------------

BasePortValue              dd     0
PortOnCommandLine          dd     0


    .
    .
    .


OSDATA ends
```

Example (continued)

```
DriverInit  proc


   CPush
   mov    DriverStackPointer, esp

   or     KeywordTextLenTable, T_REQUIRED
   lea    esi, DriverParameterBlock
   call   MSMParseCustomKeywords
   jnz    DriverInitError           ;keyword not entered


   mov    edx, BasePortValue

   (read I/O port information into eax to determine if memory
    below 16 meg is required or not)

   mov    DriverNeedsBelow16Meg, 0   ;assume below 16 not required
   or     eax, eax                   ;check if below 16 required?
   je     DriverInitRegisterHSM      ;jump if not
   mov    DriverNeedsBelow16Meg, -1  ;set below 16 flag


DriverInitRegisterHSM:

   lea    esi, DriverParameterBlock
   call   <TSM>RegisterHSM


;* Clear T_REQUIRED bit for the custom keyword so MSMParseDriverParameters will
;* not prompt for it again if it was not on the origianl command-line.

   and    KeywordTextLenTable, NOT T_REQUIRED


;* We need to set the NeedsIOPort0Bit if "PORT=" is already on the command-line.
;* Otherwise the OS will complain that it saw a standard keyword that wasn't needed.

   mov    eax, NeedInterrupt0Bit OR CAN_SET_NODE_ADDRESS
   cmp    PortOnCommandLine, 0
   je     DriverInitParse
   or     eax, NeedsIOPort0Bit


DriverInitParse:

   lea    ecx,AdapterOptions
   call   MSMParseDriverParameters
   jnz    DriverInitError

   mov    eax, BasePortValue                ;force IO Port to what
   mov    [ebx].MLIDIOPortsAndLengths, ax   ;we got from custom keyword

   call   MSMRegisterHardwareOptions
   .
   .
   .
DriverInit   endp



PortKeywordRoutine   proc

   mov    BasePortValue, eax
   mov    PortOnCommandLine, edx

PortKeywordRoutine   endp
```

MSMParseDriverParameters

On Entry

EAX	is the DriverNeedsBitMask
ECX	Pointer to DriverAdapterOptions structure
Interrupts	can be in any state
Call	at initialization time

On Return

Zero Flag	Set if successful; otherwise an error occurred.
EAX	Zero if successful; otherwise EAX points to an error message which the driver must print using *MSMPrintString* before returning to the operating system with EAX non-zero.
EBX	Pointer to the Frame Data Space
Interrupts	are disabled
Note	no registers are preserved

Description

This routine is used in conjunction with *MSMRegisterHardwareOptions* to parse the command line options.

Each standard load option corresponds to a field in the driver's configuration table. Using the *DriverNeedsBitMask* as a guide, this function collects the necessary information from the command line and from the Adapter Options Structure and fills out the appropriate fields of the configuration table.

The following pages describe the format the Adapter Options Structure and the *DriverNeedsBitMask*.

Note: During this routine the HSM's custom keywords are also processed (see "Driver Keywords" in Chapter 3)

Adapter Options

The Adapter Options Structure is defined in the ODI.INC file and is shown below. Each field of the structure is a pointer to a list of possible options for that field. If an option is not supported, a zero is placed in that field. The options correspond to fields in the driver's configuration table.

```
AdapterOptionDefinitionStructure struc

   IOSlot          dd ?     ; Ptr to a list of possible slots
   IOPort0         dd ?     ;      "  "   primary ports
   IOLength0       dd ?     ;      "  "   number of primary ports
   IOPort1         dd ?     ;      "  "   secondary ports
   IOLength1       dd ?     ;      "  "   number of secondary ports
   MemoryDecode0   dd ?     ;      "  "   primary memory values
   MemoryLength0   dd ?     ;      "  "   primary memory sizes
   MemoryDecode1   dd ?     ;      "  "   secondary memory values
   MemoryLength1   dd ?     ;      "  "   secondary memory sizes
   Interrupt0      dd ?     ;      "  "   primary interrupt values
   Interrupt1      dd ?     ;      "  "   secondary interrupt values
   DMA0            dd ?     ;      "  "   primary DMA values
   DMA1            dd ?     ;      "  "   secondary DMA values
   Channel         dd ?     ;      "  "   channel # for multichannel adapters

AdapterOptionDefinitionStructure ends
```

All lists pointed to must begin with a dword value indicating the number of options in the list. For example, the lists for an adapter with options for interrupt and port number might appear as follows.

```
IOPortOptions  dd 4                     ; number of options
               dd 300h,310h,320h,330h   ; options

IntOptions     dd 3                     ; number of options
               dd 2, 3, 5               ; options


DriverAdapterOptions  AdapterOptionDefinitionStructure
      <0,IOPortOptions,0,0,0,0,0,0,0,IntOptions,0,0,0>
```

Needs Options

The *DriverNeedsBitMask* is used to inform the parser which configuration options the driver requires.

If there are multiple possibilities for a configuration option and a driver wants this function to return which option to use, it must set the appropriate bit of the mask.

If there is only one value for a configuration option, the HSM does not set its bit in the *DriverNeedsBitMask*. The value can be set directly in the configuration table.

Equates for the bit positions of each option are provided in the ODI.INC file. These options are described in the following table.

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

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

Command Line Examples

Option	Command Line	Description
IOSlot	load <driver> SLOT=4	Use slot 4
IOPort0	load <driver> PORT=300	Base Port0 = 300h
IOLength0	load <driver> PORT=300:A	Length0 = 0Ah
IOPort1	load <driver> PORT1=700	Base Port1 = 700h
IOLength1	load <driver> PORT1=700:14	Length1 = 14h
MemoryDecode0	load <driver> MEM=C0000	Base Memory0 = C0000h
MemoryLength0	load <driver> MEM=C0000:1000	MemLength0 = 1000h (4K)
MemoryDecode1	load <driver> MEM1=CC000	Base Memory1 = CC000h
MemoryLength1	load <driver> MEM1=CC000:2000	MemLength1 = 2000h (8K)
Interrupt0	load <driver> INT=3	Interrupt0 = 3
Interrupt1	load <driver> INT1=5	Interrupt1 = 5
DMA0	load <driver> DMA=0	DMA0 = 0
DMA1	load <driver> DMA1=3	DMA1 = 3
Channel	load <driver> CHANNEL=2	Use Channel 2

Example

```
IOPortOptions  dd 4                        ; number of options
               dd 300h,310h,320h,330h      ; options

IntOptions     dd 3                        ; number of options
               dd 2, 3, 5                  ; options



DriverAdapterOptions  AdapterOptionDefinitionStructure
       <0,IOPortOptions,0,0,0,0,0,0,0,IntOptions,0,0,0>



DriverInit  proc
   •
   •
   •
  mov    eax, NeedsIOPort0Bit OR NeedsInterrupt0Bit OR CAN_SET_NODE_ADDRESS
  lea    ecx, DriverAdapterOptions
  call   MSMParseDriverParameters
  jnz    ParseParameterError
  call   MSMRegisterHardwareOptions
   •
   •
   •
```

MSMPrintString

On Entry

ECX	Possible argument #1
EDX	Possible argument #2
ESI	Pointer to a null terminated message
Interrupts	can be in any state but might be disabled during the call
Call	at initialization time only

On Return

Interrupts	are in the same state as when this routine was called
Note	EBX, EBP, EDI, and ESI are preserved

Description

This function prints the message pointed to by ESI. The HSM's initialization routine must call *<TSM>RegisterHSM* prior to using this print procedure.

The "Possible Arguments #1 and #2" above are used here the same way in which they are used in the printf routine in "C." If there are no format specifications in the string, ECX and EDX are ignored.

This routine has added functionality which supports an additional string format. If the string is preceded by a word size error number in the range of 100-999, the MSM will print the driver name, the platform name (NW for NetWare 386), and the decimal error number, before printing the specified string. (See Appendix H for a listing of standard messages.)

Example

```
ErrorMessage   dw   102
               db  "Board failed to execute reset command.",0
•
•
•
lea   ESI, ErrorMessage
call  MSMPrintString
```

The example above would output the following message if the adapter is an NE2000:

```
NE2000-NW-102: Board failed to execute reset command.
```

MSMPrintStringFatal

On Entry

ECX	Possible argument #1
EDX	Possible argument #2
ESI	Pointer to a null terminated error message
Interrupts	can be in any state but might be disabled during the call
Call	at initialization time only

On Return

Interrupts	are in the same state as when this routine was called
Note	EBX, EBP, EDI, and ESI are preserved

Description

This function prints "FATAL: " followed by the specified error message. The HSM's initialization routine must call *<TSM>RegisterHSM* prior to using this print procedure.

The "Possible Arguments #1 and #2" above are used here the same way in which they are used in the printf routine in "C." If there are no format specifications in the string, ECX and EDX are ignored. (See Appendix H for a listing of standard messages.)

Example

```
ErrorMessage   db   'Adapter %d, Error Code: %x', CR,LF,0
.
.
.
mov   ECX, BoardNumber           ; argument #1
mov   EDX, ErrorNumber           ; argument #2
mov   ESI, offset ErrorMessage
call  MSMPrintStringFatal
```

MSMPrintStringWarning

On Entry

ECX	Possible argument #1
EDX	Possible argument #2
ESI	Pointer to a null terminated error message
Interrupts	can be in any state but might be disabled during the call
Call	at initialization time only

On Return

Interrupts	are in the same state as when this routine was called
Note	EBX, EBP, EDI, and ESI are preserved

Description

This function prints "WARNING: " followed by the specified error message pointed to by ESI. The HSM's initialization routine must call *<TSM>RegisterHSM* prior to using this print procedure.

The "Possible Arguments #1 and #2" above are used here the same way in which they are used in the printf routine in "C." If there are no format specifications in the string, ECX and EDX are ignored. (See Appendix H for a listing of standard messages.)

Example

```
ErrorMessage   db   'Adapter %d, Error Code: %x', CR,LF,0
.
.
.
mov   ECX, BoardNumber           ; argument #1
mov   EDX, ErrorNumber           ; argument #2
mov   ESI, offset ErrorMessage
call  MSMPrintStringWarning
```

MSMPSemaphore (macro)

Macro Parameters

Semaphore	dword offset

On Entry

Interrupts	can be in any state
Execute	at process time only

On Return

Interrupts	are unchanged
Note	EBX, EBP, ESI and EDI are preserved

Description

MSMPSemaphore locks the real mode workspace when making an EISA BIOS call. The HSM's process might be blocked if another process has previously locked the semaphore. Once the call returns, the HSM can safely use *MSMRealModeInterrupt* to execute BIOS calls. After the HSM is done accessing the real mode interrupts, it must use *MSMVSemaphore* to allow other processes to access them.

Caution: This macro should not be used to handle critical sections that are local to the HSM.

Example

(see example for the macro *MSMGetRealModeWorkspace*)

MSMReadEISAConfig

On Entry

CH	Configuration Block Number
CL	Slot
Interrupts	may be in any state
Call	at process time only

On Return

EAX	Zero if successful; otherwise EAX contains a value indicating the results of the attempted operation. These values and their meanings are listed in the Description section below.
ESI	Pointer to the buffer containing the configuration read.
Zero Flag	Set if successful; otherwise an error occurred.
Interrupts	are unchanged
Note	EBX, ECX, and EBP are preserved

Description

MSMReadEISAConfig reads the EISA configuration block specified in CH for the slot specified in CL into a 320-byte buffer (see EISA specification). On return, EAX contains a non-zero value if the read fails for any of the following reasons:

01h - Int 15h vector removed
80h - Invalid slot number
81h - Invalid function number
82h - Nonvolatile memory corrupt
83h - Empty slot
86h - Invalid BIOS routine call
87h - Invalid system configuration

Note: The information returned should be copied into local memory. Once the driver returns to the operating system or calls a blocking routine the information in the buffer may change.

Example

```
DriverInit  proc
      :
      :                                  ;ebx = ptr to the Frame Data Space
   movzx ecx, [ebx].MLIDSlot             ;start with block 0 and correct slot

ReadConfigBlockLoop:
   call  MSMReadEISAConfig               ;get configuration block
   jnz   ReadEISAConfigError             ;jump if error
   inc   ch                              ;set ch to next config block
   test  BYTE PTR [esi+n], Valid_Data    ;does buffer contain desired data
   jz    ReadConfigBlockLoop             ;try next config block
      :
      :
```

MSMReadPhysicalMemory

On Entry

ECX	Number of bytes to read
ESI	physical source address (where to read data from)
EDI	logical destination address (where to transfer data to)
Interrupts	may be in any state
Call	during *DriverInit* before *MSMRegisterHardwareOptions*

On Return

Note	EBX, EBP, ESI, and EDI are preserved

Description

If the driver attempts to access shared RAM before calling *MSMRegisterHardwareOptions,* a page fault abend will occur on the server. Accesses to the shared RAM prior to registration do not normally happen unless the HSM must obtain additional information such as interrupt numbers or shared RAM buffer size for the configuration table.

This routine can be used to read information from a shared RAM physical address before hardware registration.

See also *MSMWritePhysicalMemory*

Example

```
mov   esi, SourceAddress         ; physical shared RAM address source
lea   edi, [ebx].MLIDInterrupt   ; logical dest. in frame data space
mov   ecx, 1                     ; read 1 byte

call  MSMReadPhysicalMemory         ; transfer data
cmp   eax, 0                        ; check for errors
jne   ErrorReadingFromSharedMemory  ; Jump if so
```

MSMRealModeInterrupt (macro)

Macro Parameters

InputStructure	19 byte Register structure
OutputStructure	20 byte Register structure

On Entry

Interrupts	can be in any state
Execute	at process time only

On Return

EAX	Zero if successful; otherwise the interrupt vector was unavailable because DOS has been removed.
Zero Flag	Set if successful
Interrupts	are preserved on return, but may have been changed during the call.
Note	EBX, EBP, ESI and EDI are preserved

Description

MSMRealModeInterrupt performs real mode interrupts, such as BIOS and DOS interrupts. EISA boards must use *MSMRealModeInterrupt* to perform the INT 15h BIOS call that returns the board configuration.

This process might relinquish control to other procedures during execution.

Example

(see example for the macro *MSMGetRealModeWorkspace*)

MSMRegisterHardwareOptions

On Entry

Interrupts	can be in any state
Call	at initialization time only

On Return

EAX = 0 EAX = 1 EAX = 2 EAX > 2	New Adapter was successfully registered New Frame Type was successfully registered New Channel (multichannel adapters) was registered Pointer to an error message. (hardware registration failed)
EBP	Pointer to the Adapter Data Space if successful
EBX	Pointer to the Frame Data Space if successful
Interrupts	are preserved

Description

This function must be called by the HSM's *DriverInit* routine to register the hardware options.

On return from *MSMRegisterHardwareOptions*:

If EAX is 0, a new adapter was registered and the driver should continue with initializing the adapter. If a new adapter is being added, the memory associated with the Adapter Data Space is allocated and control returns to *DriverInit* with EBP pointing to that space.

If EAX is 1, a new frame type was registered for an existing adapter and the *DriverInit* routine is basically finished.

If EAX is 2, a new channel was registered for an existing multichannel adapter. The driver (and MSM) typically treat the registering of a new channel as a new adapter.

If EAX is > 2, the MSM was unable to register the hardware options (typically due to conflicts with existing hardware). In this case, EAX points to an error message which the driver should print using *MSMPrintString*. *DriverInit* should then return immediately to the operating system with EAX set to any non-zero value.

Example

```
DriverInit  proc
     ⋮
   call  MSMParseDriverParameters

   call  MSMRegisterHardwareOptions

   cmp   eax,2
   ja    DriverInitError
   je    NewChannel

   cmp   eax,1
   je    NewFrame

  ;(Initialize for NewAdapter)

     ⋮


DriverInitExit:
   xor   eax,eax
   ret

DriverInitError:
   mov   esi,eax
   call  MSMPrintString
   or    eax,-1
   ret

DriverInit  endp
```

MSMRegisterMLID

On Entry

EBP	Pointer to the Adapter Data Space
EBX	Pointer to the Frame Data Space
Interrupts	may be in any state
Call	at process time only

On Return

EAX	Zero if successful; otherwise EAX points to an error message which the driver must print using *MSMPrintString* before returning to the operating system with EAX non-zero.
Zero Flag	Set if successful; otherwise an error occurred.
Interrupts	are unchanged
Note	EBX and EBP are preserved

Description

After *DriverInit* has successfully initialized the adapter, it should call this routine to register the MLID with the Link Support Layer.

Note: When this routine returns, the configuration table contains a valid board number. HSMs for intelligent bus master adapters may now pass the board number and frame ID information to the adapter if necessary.

Example

```
DriverInit  proc
      ⋮
   call  MSMRegisterMLID
   jnz   RegisterMLIDError
      ⋮
```

MSMRescheduleLast (macro)

On Entry

Interrupts	can be in any state (but might be enabled during the call)
Execute	at process time only

On Return

Interrupts	are in the same state as when the routine was called
Note	EBX, EBP, ESI, and EDI are preserved

Description

MSMRescheduleLast places the task last on the list of active tasks to be executed. This routine must be called only at process time because it suspends the process and could change the machine state. *MSMRescheduleLast* should be used only in the driver initialization and driver remove procedures.

The example below illustrates the NE3200 driver using it to acquire the HSM's first RCB during the driver initialization procedure. Because the NIC is a bus-master adapter, the HSM must have at least one RCB to start. This means that the HSM must let other processes execute until an RCB is available.

Example

```
    sti                           ; Enable Real Time Clock
    MSMGetCurrentTime             ; EAX = Time
    lea  ecx, [eax].(5 * 20)      ; ECX = Time + ~ 5 seconds

GetFirstRCBLoop:

    mov  esi, CommonMaximumSize   ; ESI = Max Packet Size
    call MSMAllocateRCB           ; Allocate an RCB
    jz   short GotFirstRCB        ; Jump if successful

    MSMGetCurrentTime             ; EAX = Current Time
    cmp  eax, ecx                 ; Timed out?
    lea  eax, NoFirstRCBMsg       ; EAX -> Error Message
    jae  DriverResetErrorExit     ; Exit if so
    pushad                        ; Save all registers
    MSMRescheduleLast             ; Let other processes have
    popad                         ;  a turn to execute
    jmp  short GetFirstRCBLoop    ; Try it again

GotFirstRCB:
```

MSMReturnDriverResources

On Entry

Interrupts	are disabled
Call	at process time only

On Return

Interrupts	remain disabled
Note	All registers are destroyed

Description

If the HSM's *DriverInit* routine is unable to initialize the adapter and has already called *<TSM>RegisterHSM*, it must call this routine to return the driver's resources before exiting.

Example

```
DriverInit  proc

   Cpush
    ⋮
   call  <TSM>RegisterHSM
   jnz   DriverInitError
    ⋮
   [*** Initialize the Adapter ***]
   call  DriverReset
   jnz   DriverInitResetError
    ⋮

DriverInitResetError:
   push  eax
   call  MSMReturnDriverResources
   pop   eax
DriverInitError:
   mov   esi, eax
   call  MSMPrintString
   or    eax, 1
   Cpop
   ret

DriverInit  endp
```

MSMReturnNotificationECB (macro) MSMFastReturnNotificationECB (macro)

On Entry

ESI	Pointer to the notification ECB
Interrupts	can be in any state
Execute	at process or interrupt time

On Return

Interrupts	are disabled
Note	*MSMReturnNotificationECB* ESI, EDI, and EBP are preserved *MSMFastReturnNotificationECB* Assume all registers are destroyed

Description

Drivers that support outside management NLMs (such as HMI or CSL) use these macros to process notification ECBs containing management alert information.

If the hardware generates an alert, the HSM obtains a notification ECB using *MSMAllocateRCB*. This procedure requires a packet size on entry. The size specified will depend on the amount of information that must be passed up to the management application. The driver fills in the ECB with the notification information according to the driver management specification, sets ESI to point to the ECB, and returns the notification ECB using one of these macros.

MSMReturnNotificationECB places the ECB in the LSLs holding queue and waits for the HSM to call *MSMServiceEvents* before passing it to the management NLM. *MSMFastReturnNotificationECB* passes the ECB immediately to the management application.

Example

```
HubResetNotification  proc
   ⋮
   mov   esi, 4
   call  MSMAllocateRCB                   ; Get notification ECB
   ⋮
   ⋮  (Fill in all required notification information)
   ⋮
   mov   esi, ECBPtr                      ; Point to the ECB
   MSMFastReturnNotificationECB           ; Return the ECB directly to
   ⋮                                        the management application
```

MSMReturnRCB (macro)

On Entry

ESI	Pointer to the unneeded RCB
Interrupts	can be in any state
Execute	at process or interrupt time

On Return

Interrupts	are disabled
Note	EBX, ECX, EDX, EBP, and EDI are preserved

Description

MSMReturnRCB returns an unneeded RCB to the LSL. This routine is called to discard the RCB, not to process it. To return an RCB for processing, see *<TSM>RcvComplete* or *<TSM>ProcessGetRCB*.

Example

```
    mov  esi, [ebp].ReceiveQueueHead      ; ESI -> First RCB
    mov  [ebp].ReceiveQueueHead, 0        ; Clear pointer
    or   esi,esi                          ; Valid RCB?
    jz   ShutdownAllRCBsReturned          ; Jump if not

ShutdownReturnRCBLoop:

    mov  ecx, [esi].RCBDriverWS+4         ; ECX -> Next RCB
    MSMReturnRCB                          ; Return RCB
    mov  esi, ecx                         ; ESI -> Next RCB
    or   esi, esi                         ; Valid RCB?
    jnz  ShutdownReturnRCBLoop            ; Jump if so
```

MSMScheduleAESCallBack

On Entry

EBP	Pointer to the Adapter Data Space
EAX	Time Interval in ticks (1 tick ≈ 1/18 sec)
Interrupts	can be in any state, but are disabled during the call
Call	only at initialization time (during *DriverInit*)

On Return

EAX	Zero if successful; otherwise EAX points to an error message which the driver must print using *MSMPrintString* before returning to the operating system with EAX non-zero.
Zero Flag	Set if successful; otherwise an error occurred.
Interrupts	are preserved
Note	EBX and EBP are preserved

Description

This routine can be called during *DriverInit* to enable a periodic call back to the HSM's *DriverAESCallBack* routine. Once enabled, *DriverAESCallBack* is invoked during process time at the intervals specified by EAX. The MSM sets up the Adapter and Frame Data Space before calling *DriverAESCallBack* and automatically schedules a new call back on return.

Note: *DriverAESCallBack* is used if any calls are made to routines which can be invoked at process time only. *DriverINTCallBack* should be used instead of *DriverAESCallBack* when possible. (see *MSMScheduleIntTimeCallBack*)

Example

```
DriverInit  proc
      ⋮
   mov   eax, 18                       ; Schedule call back in 18 ticks
   call  MSMScheduleAESCallBack
   jnz   ScheduleCallBackError
      ⋮
```

MSMScheduleIntTimeCallBack

On Entry

EBP	Pointer to the Adapter Data Space
EAX	Time Interval in ticks (1 tick ≈ 1/18 sec)
Interrupts	are disabled and remain disabled
Call	only at initialization time (during *DriverInit*)

On Return

EAX	Zero if successful; otherwise EAX points to an error message which the driver must print using *MSMPrintString* before returning to the operating system with EAX non-zero.
Zero Flag	Set if successful; otherwise an error occurred.
Interrupts	are disabled
Note	EBX and EBP are preserved

Description

This routine can be called during *DriverInit* to enable a periodic call back to the HSM's *DriverINTCallBack* routine. Once enabled, *DriverINTCallBack* is invoked during the timertick interrupt at the interval specified by EAX. The MSM sets up the Adapter and Frame Data Space before calling *DriverINTCallBack* and automatically schedules a new call back on return.

Note: *DriverINTCallBack* cannot be used if calls are made to routines which can be invoked only at process time. *DriverAESCallBack* should be used instead. (see *MSMScheduleAESCallBack*)

Example

```
DriverInit  proc
      ⋮
   mov   eax, 18                      ; Schedule call back in 18 ticks
   call  MSMScheduleIntTimeCallBack
   jnz   ScheduleCallBackError
      ⋮
```

MSMServiceEvents (macro)

On Entry

Interrupts	can be in any state
Execute	at process or interrupt time

On Return

Interrupts	are disabled on completion, but might have been enabled during execution
Note	all registers are destroyed

Description

If the HSM has used *<TSM>SendComplete, <TSM>RcvComplete* or *<TSM>ProcessGetRCB,* it must use either *MSMServiceEvents* or *MSMServiceEventsAndRet* before it exits back to the operating system.

If the HSM must execute any instructions after it services events, then it must use *MSMServiceEvents* instead of *MSMServiceEventsAndRet.*

In the example below, the adapter supports shared interrupts. In this case, the operating system requires that EAX equal 0 if the interrupt is for the HSM. The HSM must use *MSMServiceEvents* and set EAX to 0 before returning. If *MSMServiceEventsAndRet* is used, the HSM returns before it is able to set EAX to 0. If the HSM does not support shared interrupts, it can return immediately after servicing events, therefore, the *MSMServiceEventsAndRet* macro should be used.

Note: If the HSM uses *<TSM>FastSendComplete, <TSM>FastRcvComplete,* or *<TSM>FastProcessGetRCB* exclusively, it does not need to use *MSMServiceEvents.* The "fast" routines service events before returning.

Example

```
DriverISR   proc
     ⋮

DriverISRExit:

    MSMServiceEvents    ; Service Events queue
    xor  eax, eax       ; Inform operating system that interrupt was ours
    ret

DriverISR   endp
```

MSMServiceEventsAndRet (macro)

On Entry

Interrupts	can be in any state
Execute	at process or interrupt time

On Return

Note	this macro does not return to the HSM

Description

If the HSM has used *<TSM>SendComplete, <TSM>RcvComplete,* or *<TSM>ProcessGetRCB*, it must use either *MSMServiceEvents* or *MSMServiceEventsAndRet* before it exits back to the operating system.

Since this macro automatically returns, *MSMServiceEventsAndRet* must be the last executable line of code in the routine. If the HSM must execute any instructions after servicing events, it must use the *MSMServiceEvents* macro which does not automatically return.

Note: If the HSM uses *<TSM>FastSendComplete, <TSM>FastRcvComplete,* or *<TSM>FastProcessGetRCB* exclusively, it does not need to use *MSMServiceEvents.* The "fast" routines service events before returning.

Example

```
DriverISR   proc
     ⋮

DriverISRExit:

    MSMServiceEventsAndRet    ; Service Events and Return.

DriverISR   endp
```

MSMSetHardwareInterrupt

On Entry

EBP	Pointer to the Adapter Data Space
EBX	Pointer to the Frame Data Space
Interrupts	are disabled and remain disabled
Call	at process time

On Return

EAX	Zero if successful; otherwise EAX points to an error message which the driver must print using *MSMPrintString* before returning to the operating system with EAX non-zero.
Zero Flag	Set if successful; otherwise an error occurred.
Interrupts	are disabled
Note	EBX and EBP are preserved

Description

The HSM's *DriverInit* routine will call this function to set up a hardware interrupt.

Example

```
call  MSMRegisterHardwareOptions
call  MSMSetHardwareInterrupt
jnz   SetHardwareIntError
```

MSMStartCriticalSection (macro)

On Entry

EBP	Pointer to the Adapter Data Space
Interrupts	can be in any state
Execute	at process or interrupt time

On Return

Interrupts	are unchanged
Note	all registers are preserved

Description

The *MSMStartCriticalSection* and *MSMEndCriticalSection* macros are used to prevent the TSM from calling *DriverSend* while it is performing critical operations. This allows interrupts to be enabled in the *DriverSend* and/or *DriverISR* routine.

When a TCB needs to be sent, the TSM usually calls *DriverSend*. However, *DriverSend* may be reading bytes from the card and starting a send at this point could corrupt data. If the HSM is in a critical section, the TSM queues the packet instead of calling *DriverSend*.

Note: Critical sections can be nested.

Example

(see example for the macro *MSMEndCriticalSection*)

MSMVSemaphore (macro)

Macro Parameters

Semaphore	dword offset

On Entry

Interrupts	can be in any state
Execute	at process or interrupt time

On Return

Interrupts	are unchanged
Note	EBX, EBP, ESI and EDI are preserved

Description

The *MSMVSemaphore* macro clears a semaphore that was set with *MSMPSemaphore*. *MSMVSemaphore* is usually used when the HSM has finished making an EISA BIOS call to allow other processes to use the workspace.

Example

(see example for the macro *MSMGetRealModeWorkspace*)

MSMWritePhysicalMemory

On Entry

ECX	Number of bytes to write
ESI	logical source address (where to read data from)
EDI	physical destination address (where to transfer data to)
Interrupts	may be in any state
Call	during *DriverInit* before *MSMRegisterHardwareOptions*

On Return

Note	EBX, EBP, ESI, and EDI are preserved

Description

If the driver attempts to access shared RAM before calling *MSMRegisterHardwareOptions,* a page fault abend will occur on the server. Accesses to the shared RAM prior to registration do not normally happen unless the HSM must obtain additional information such as interrupt numbers or shared RAM buffer size for the configuration table.

This routine can be used to write information to a shared RAM physical address before hardware registration.

See also *MSMReadPhysicalMemory*

Example

```
    mov   edi, DestinationAddress    ; physical shared RAM address
    lea   esi, [ebx].MLIDNodeAddress ; logical source is in frame data space
    mov   ecx, 6                     ; write 6 byte node address

    call  MSMWritePhysicalMemory     ; transfer data
    cmp   eax, 0                     ; check for errors
    jne   ErrorWritingToSharedMemory ; Jump if so
```