5

HSM Procedures

Introduction

This chapter describes the routines that are the primary components of the Hardware Specific Module (HSM).

Initialization and Removal
- *DriverInit* *(required)*
- *DriverRemove* *(required)*

Board Service
- *DriverISR*
- *DriverPoll*

(one required)

Packet Transmission
- *DriverSend* *(required)*

Multi-Operating System Support
- *DriverEnableInterrupt* *(recommended)*
- *DriverDisableInterrupt* *(recommended)*

Control Procedures
- *DriverReset* *(required)*
- *DriverShutdown* *(required)*
- *DriverMulticastChange* *(recommended)*
- *DriverPromiscuousChange* *(recommended)*
- *DriverStatisticsChange* *(optional)*
- *DriverRxLookAheadChange* *(optional)*
- *DriverManagement* *(optional)*

Timeout Detection
- *DriverAESCallBack* *(optional)*
- *DriverINTCallBack* *(optional)*
- *DriverTxTimeout* *(RX-Net drivers only)*

Every driver must provide the *required* procedures in order to function properly. The *recommended* procedures should be implemented if the hardware supports that function. The *optional* procedures are available if the adapter or driver requires the functionality. The HSM indicates routines not supported by placing a zero in the corresponding fields of the *DriverParameterBlock*.

All procedures described on the following pages are near calls from the MSM and TSM. The pseudocode shown is intended to illustrate a general flow of events and does not necessarily describe optimized code. Refer to Appendix I for a sample server driver template.

Initialization

The HSM's *DriverInit* routine controls the complete initialization process, although specific tasks performed during initialization are handled by MSM or TSM routines. The initialization tasks include:

- Allocate the Frame and Adapter Data Space
- Process custom command line keywords and custom firmware
- Parse the standard LOAD command line options
- Register hardware options
- Initialize the adapter hardware
- Register the driver with the Link Support Layer

This section explains how the initialization tasks are divided between the HSM and the support modules. Following the discussion is pseudocode for a *DriverInit* routine.

DriverInit

When the NetWare OS receives the command to load the driver, it calls the *DriverInit* routine (specified as the **"start"** routine in HSM's linker definition file). *DriverInit* must preserve EBP, EBX, ESI, and EDI on the stack, and set the *DriverStackPointer* field of the *DriverParameter-Block* to the value of ESP. The HSM then registers with the MSM and TSM interface as described in the next section.

Register with the MSM / TSM

DriverInit calls the *<TSM>RegisterHSM* routine with ESI pointing to the *DriverParameterBlock*. The TSM passes the driver's parameter block pointer along with its own to the MSM.

The MSM makes a local copy of both parameter blocks and processes the information passed on the stack from the operating system. If the HSM has custom firmware, the MSM loads the firmware and initializes the *DriverFirmwareSize* and *DriverFirmwareBuffer* variables as described in Chapter 3.

The MSM allocates memory for the Frame Data Space and creates a copy of the driver's configuration table template in that area. If the *MLIDCardName* and *MLIDMajorVersion* fields of the configuration table are initialized to zero, the MSM fills in these fields and the *MLIDMinorVersion* field using information derived from the linker definition file. If the HSM has placed non-zero values in the card name and major version fields, these fields are not modified.

Finally, the MSM sets the *MLIDMaximumSize* field of the configuration table to the LSL's maximum packet size and returns to *DriverInit*.

- If the MSM was unsuccessful in its initialization tasks, it returns with EAX pointing to an error message. *DriverInit* should print the message using *MSMPrintString* and return to the operating system with EAX set to a non-zero value.

- If the MSM is successful, it returns with EAX set to zero and EBX pointing to the driver's configuration table in the Frame Data Space. The HSM must now gather the hardware option information needed for the configuration table and call the MSM to parse the driver parameters entered on the command line. This process is described in the following section.

Determine Hardware Options

After *<TSM>RegisterHSM* returns successfully, the driver must determine the hardware configuration of the adapter. This includes parameters such as the slot number for MCA or EISA adapters, the base port for programmed IO adapters, memory decode addresses for shared RAM adapters, interrupt numbers, and DMA channels. In MCA or EISA machines, the driver can obtain this information directly from the system once the slot number has been identified.

DriverInit should perform each of the following steps if appropriate for the hardware.

1) If the HSM supports both ISA and MCA buses, it should use the *MSMGetHardwareBusType* macro to determine the bus type.

2) If the HSM supports an EISA or MCA bus, scan all slots to search for the adapter's ID. Any slots that are found should be recorded in the IOSlot option list of the *AdapterOptionDefinitionStructure*. This structure is described in Chapter 7 under the *MSMParseDriver-Parameters* routine.

3) The HSM next calls *MSMParseDriverParameters* to determine the hardware configuration options (or slot number) specified on the load command line and to query the operator for any required parameters which were not specified.

   The *MSMParseDriverParameters* procedure requires an *Adapter-OptionDefinitionStructure* containing the valid options for the hardware configuration. A *NeedsBitMap* is also required to indicate which specific hardware options must be obtained either from the command line or from the console operator.

The table below shows the correspondence between the load options and configuration table fields. The standard load command options are described in Appendix A. An example load command is shown here:

```
load <driver> frame=ethernet_802.3, port=300, int=3
```

Configuration Table Fields	Command Line
MLIDSlot	load <driver> SLOT=4
MLIDIOPort0	load <driver> PORT=300
MLIDIORange0	load <driver> PORT=300:A
MLIDIOPort1	load <driver> PORT1=700
MLIDIORange1	load <driver> PORT1=700:14
MLIDMemoryDecode0	load <driver> MEM=C0000
MLIDMemoryLength0	load <driver> MEM=C0000:1000
MLIDMemoryDecode1	load <driver> MEM1=CC000
MLIDMemoryLength1	load <driver> MEM1=CC000:2000
MLIDInterrupt0	load <driver> INT=3
MLIDInterrupt1	load <driver> INT1=5
MLIDDMAUsage0	load <driver> DMA=0
MLIDDMAUsage1	load <driver> DMA1=3
MLIDChannelNumber	load <driver> CHANNEL=2

MSMParseDriverParameters also processes any custom command line keywords defined by the *DriverKeyword* variables in the *DriverParameterBlock*. (see also *MSMParseCustomKeywords*)

On return from *MSMParseDriverParameters*, the I/O portion of the logical board's configuration table in the Frame Data Space has been filled in with the parsed values.

4) For EISA or MCA buses, the configuration table now contains the selected adapter slot number. If the adapter is on an MCA bus, the HSM should read the appropriate POS registers for the slot to determine the configuration. For EISA adapters, the HSM should call *MSMReadEISAConfig* to determine the configuration.

When all needed information has been obtained for the configuration table, *DriverInit* calls *MSMRegisterHardwareOptions* which is described in the next section.

Note: If the driver must access shared memory before registering the hardware options, it must use *MSMReadPhysicalMemory* and *MSMWritePhysicalMemory*.

Register Hardware Options

The HSM calls *MSMRegisterHardwareOptions* to register with the operating system. This routine reports to the HSM whether a new adapter or a new frame format for an existing adapter is being loaded. If a new adapter is being registered, the MSM allocates the Adapter Data Space and copies the driver's *AdapterDataSpaceTemplate* to that area. This routine also notifies the HSM of any conflicts with existing hardware in the system.

There are four possible conditions that the HSM must handle on return from *MSMRegisterHardwareOptions*:

- If EAX = 0, a new adapter was successfully registered and the HSM must proceed with the hardware initialization (EBP now contains a pointer to the Adapter Data Space).

- If EAX = 1, a new frame type for an existing adapter was successfully registered and initialization is essentially complete.

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

- If EAX > 2, the MSM was unable to register the hardware options and EAX points to an error message. *DriverInit* must print the error message using *MSMPrintString* and return to the operating system with EAX set to a non-zero value.

Setup a Board Service Routine

The HSM registers its board service routine, *DriverISR* or *DriverPoll*, by calling either *MSMSetHardwareInterrupt* or *MSMEnablePolling*. The *DriverISR* description later in this chapter provides special instructions on setting up and handling shared interrupts.

Initialize the Adapter

At this point the HSM initializes the adapter hardware. This consists of all setup appropriate for the hardware and might also include RAM and other hardware tests. The *DriverReset* routine could be called to handle part of this procedure.

Note: It is important that *DriverInit* sets up the correct number of transmit buffers (the maximum number of simultaneous sends allowed by the hardware) by placing an appropriate value in *MSMTxFreeCount*. A description of this variable is in Chapter 4 and information about its use is in the packet transmission section of this chapter.

If an error occurs during the hardware initialization, *DriverInit* should print an appropriate error message, call *MSMReturnDriverResources*, and return to the operating system with EAX set to a non-zero value. If the hardware initializes successfully, the HSM then registers the driver with the LSL.

Register with the LSL

DriverInit calls the *MSMRegisterMLID* routine to register the driver with the Link Support Layer. Registration consists of the MSM passing the addresses of the MSM's send and control handler procedures, and a pointer to the HSM's configuration table to the LSL. The LSL assigns a logical board number to the adapter and the MSM places it in the configuration table. The MSM automatically registers a logical board with the LSL during *MSMRegisterHardwareOptions* each time a new frame is added for an existing adapter. If an error occurs, the MSM routine returns a pointer to an error message in EAX.

If *MSMRegisterMLID* is successful, 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.

Schedule Timeout Callbacks

If the HSM is running an interrupt driven adapter, it may need to schedule a timer event that checks to see if the board was unable to complete a send. To establish this timer event, the driver uses either *MSMScheduleIntTimeCallBack* or *MSMScheduleAESCallBack*. These routines schedule periodic calls to the HSM's *DriverCallBack* or *DriverAES* routines. (RX-Net drivers normally use *DriverTxTimeout*, but could use these other two routines.)

If the adapter is not interrupt driven, the polling procedure can check to see if it failed to complete a send.

DriverInit Pseudocode

On Entry

Interrupts	are enabled
Note	this routine executes at a process level

On Return

EAX	zero if successful; non-zero if an error occurred
Interrupts	may be in any state

```
DriverInit   proc

   push ebx, ebp, esi, edi
   mov  DriverStackPointer, esp
   lea  esi, DriverParameterBlock
   call <TSM>RegisterHSM
   jnz  DriverInitError

*** Determine Hardware Options ***

   If using an MCA or EISA Adapter
           Search all slots for the Adapter's ID
           Build an IOSlot list for the AdapterOptionStructure

   call  MSMParseDriverParameters
   jnz   DriverInitError

   If using an MCA or EISA Adapter
           Use MLIDSlot value to obtain configuration information...
               If MCA:  ...from POS registers
               If EISA:  ...from configuration block using MSMReadEISAConfig

*** Register Hardware Options ***

   call  MSMRegisterHardwareOptions
   If an Error occurred
           jmp    DriverInitError
   else if a New Frame was added
           jmp    DriverInitExit
   else a New Adapter was registered
           continue with full initialization

*** Setup a Board Service Routine ***

   call  MSMSetHardwareInterrupt  (or MSMEnablePolling)
   jnz   DriverInitError
```

-(continued)-

```
*** Initialize the Adapter ***

   If there is not a Node Address override
         Read in the Node Address from the board
         Copy the Node Address to the Configuration Table

   Initialize MSMTxFreeCount

   Initialize the Adapter Hardware, etc...
   call   DriverReset to handle some tasks
   If there was an error initializing the hardware
         call  MSMReturnDriverResources
         jmp  DriverInitError

*** Register with the LSL ***

   call   MSMRegisterMLID
   jnz    DriverInitError

*** Schedule Timeout Callbacks ***

   If Timeout detection is required
         eax = callback interval in ticks
         call  MSMScheduleIntTimeCallBack  (to enable callbacks to DriverCallBack)
         - or -
         call  MSMScheduleAESCallBack      (to enable callbacks to DriverAES)
         jnz   DriverInitError


DriverInitExit:

   eax = zero   (Initialization was successful)
   pop   edi, esi, ebp, ebx
   return


DriverInitError:

   esi = eax  (Ptr to Error Message)
   call   MSMPrintString
   eax = non-zero value   (Initialization Failed)
   pop   edi, esi, ebp, ebx
   return


DriverInit   endp
```

Packet Reception

This section provides a brief overview of the commonly used reception methods available to the developer.

When the adapter receives a packet, the HSM must copy the packet into an RCB obtained from the TSM. The RCB is passed back to the TSM where it is processed and transferred to the Link Support Layer. The Link Support Layer then directs it to the proper protocol stack.

Reception Methods

The method of packet reception selected is typically dependent on the adapter's data transfer method. The examples on the following pages are intended to illustrate a general flow of events. Refer to the appropriate MSM and TSM support call descriptions for detailed information.

In general, packet reception involves the following steps:

- Obtain a Receive Control Block (RCB) structure from the TSM. RCBs may be allocated before or after a packet is received.
- Copy the packet into the RCBDataBuffer or RCBDataFragments.
- Return the RCB back to the TSM (RCBs will be placed in the LSL's holding queue until the HSM issues a service events command).
- Use the *MSMServiceEvents* macro to allow the LSL to call the transmit ECB's event service routine.

Programmed I/O and Shared RAM

Option 1. This is the simplest reception method. During development it may be helpful to initially use this method, then implement Option 2 after the HSM is functioning properly. The steps performed for this reception method are outlined below. The *<TSM>ProcessGetRCB* procedure in Chapter 6 provides a detailed description of this process.

DriverISR

Call MSMAllocateRCB to get an RCB (unless you already have one from last step)
Copy the received packet into the RCBDataBuffer.
Call <TSM>ProcessGetRCB

The TSM checks the header information and if valid:
- fills in the remainder of the RCB fields
- delivers the RCB to the LSL
- returns a new RCB to the driver

Save the new RCB for next packet received
MSMServiceEvents

Option 2. This method involves using a LookAhead process, in which the frame header information is first confirmed before the entire packet is transferred from the adapter into an RCB. For this reason, Option 2 is recommended over Option 1.

The adapter's data transfer mode determines how the LookAhead process is handled. Programmed I/O adapters must transfer *MSMMaxFrameHeaderSize* bytes into a LookAhead buffer allocated for this purpose. If the adapter uses a shared RAM transfer mode, the LookAhead buffer is simply the start of the packet in shared RAM.

The steps performed for this reception method are outlined below. The *<TSM>GetRCB* procedure in Chapter 6 provides a detailed description of this process.

DriverISR

Setup a LookAhead buffer as described above (**MSMMaxFrameHeaderSize** bytes)
Call <TSM>GetRCB (with a pointer to the LookAhead buffer in ESI)

TSM checks the header information and if valid:
- obtains an RCB
- fills in the RCBReserved fields
- returns a pointer to the RCB in ESI

Copy the remainder of the packet into the RCB fragments
Call <TSM>RcvComplete
MSMServiceEvents

DMA and Bus Master

Option 1. This reception method is used for most bus master adapters in which the RCBs are preallocated. The steps performed for this reception method are outlined below. The *<TSM>ProcessGetRCB* procedure in Chapter 6 provides a detailed description of this process.

```
DriverInit

  Use MSMAllocateRCB to obtain first RCB(s)
  Queue RCB(s) until a packet is received in DriverISR.

DriverISR

  Copy received packet into the RCBDataBuffer.
  Call <TSM>ProcessGetRCB
       The TSM checks the header information and if valid:
          • fills in the remainder of the RCB fields
          • delivers the RCB to the LSL
          • returns a new RCB to the driver

  Queue the new RCB until next packet is received
  MSMServiceEvents
```

Option 2. This method is recommended for intelligent adapters that are designed to be "ECB aware". It reduces the load on the server by off-loading code to the adapter. In this way, the adapter's firmware handles most of the reception process. The steps performed for this reception method are outlined below.

```
DriverInit

  Use MSMAllocateRCB to obtain first RCB(s)
  Queue RCB(s) until a packet is received.

Firmware

  Filters the frame header information and if valid, fills in all
  fields of the ECB as described in Chapter 4.  Generates interrupt
  when receive is complete (ready).

DriverISR

  Call <TSM>RcvComplete to return the completed RCB.
  Use MSMAllocateRCB to obtain another RCB for queue
  MSMServiceEvents
```

RX-Net

Option 1. This option is used for RX-Net shared RAM adapters. The steps performed for this reception method are outlined below. The *RXNetTSMRcvEvent* procedure in Chapter 6 provides a detailed description of this process.

```
DriverISR

  Set ESI to point to received packet.
  Call RXNetTSMRcvEvent

      The TSM copies the entire packet into an RCB if the fragment
      is wanted with no other interaction from the driver.

  MSMServiceEvents
```

Option 2. This option is used for RX-Net programmed I/O adapters. The steps performed for this reception method are outlined below. The *RXNetTSMGetRCB* procedure in Chapter 6 provides a detailed description of this process.

```
DriverISR

  Set ESI to point to a LookAhead buffer containing the
     header information as shown in Figure 5.1.
  Call RXNetTSMGetRCB

      The TSM checks the packet header information to see if the packet fragment
      is wanted and if so, returns a pointer to an RCB.

  Determine the current position in the RCB fragment buffers and copies
     the data into the RCB.
  Update the packet length field of the RCB.

  Call RXNetTSMRcvComplete
  MSMServiceEvents
```

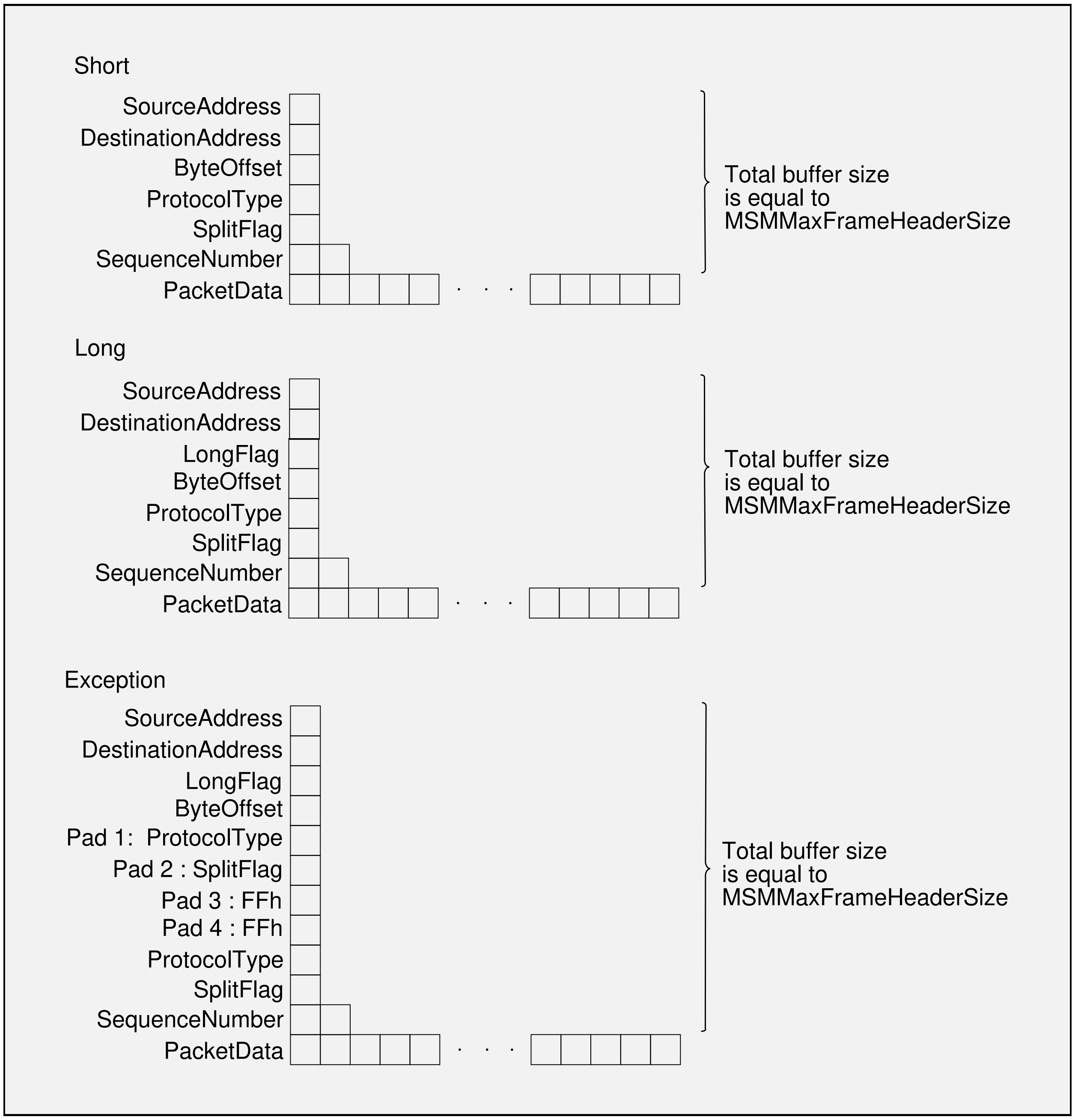

Figure 5.1 Format of RX-Net LookAhead Buffer

Board Service

The board service routine generally needs to detect and handle both receive events and transmit complete events. The driver can be notified of these events by using either an interrupt service routine, *DriverISR,* a polling procedure, *DriverPoll,* or a combination of both. These routines are explained next.

DriverISR

DriverISR is called by the MSM when a hardware interrupt is detected. The driver needs only to service the adapter and return (do not use `iret`).

Note: Novell recommends that interrupts remain unaltered during *DriverISR*. Drivers should allow the support modules to control the interrupt state via calls to the *DriverEnableInterrupt* and *DriverDisableInterrupt* routines at the appropriate times. If a driver procedure must alter the interrupt state, it must restore the interrupt state before returning.

The interrupt service routine generally needs to detect and handle the following events:

- Receive Event
- Receive Error
- Transmit Complete
- Transmit Error

The ISR routine should continue checking for receive and transmit events until there are no more to be serviced.

Error detection and handling are optional in the cases where the hardware is able to handle transmit and receive errors without driver intervention. Even if the hardware has this capability, the driver must still be able to update or maintain the statistics table described in Chapter 3.

Receive Event

The receive portion of the board service routine checks for receive errors and jumps to an error handler if an error has occurred. Otherwise, the routine services the packet using one of the reception methods described in the previous section.

Receive Error

If the HSM encounters a receive error, it should perform the following actions:

- **Attempt to identify the error.** While some cards provide greater diagnostic support than others, the HSM should attempt to pinpoint the specific cause of the error (buffer overflow, missed packet, checksum error, etc.).

- **Increment diagnostic counters.** The HSM should maintain the diagnostic counters in the statistics table for every detectable error condition. This will aid in debugging the driver as well as maintaining it in the future. The driver should also increment the generic statistic *TotalRxMiscCount* if a fatal receive error occurred that is not counted in any other standard counter. Fatal receive errors may also be counted by the TSM using a media specific counter as well.

Transmit Complete

Each time the HSM detects a successfully completed transmit event, it should perform the following functions:

- Release the TCB using (if not already released in *DriverSend*)

```
call  <TSM>SendComplete
```

- Increment the number of available transmit resources

```
inc   [ebp].MSMTxFreeCount
```

- Transmit the next packet if one is waiting to be sent

```
test  [ebp].MSMStatusFlags,TXQUEUED
jz    NoSendsQueued
call  <TSM>GetNextSend
jnz   NoSendsQueued
call  DriverSend
```

Transmit Errors

If the HSM encounters a transmit error, it should perform the following actions:

- **Attempt to identify the error.** As with receive errors, the HSM should try to pinpoint the specific cause of the error (excess collisions, cable disconnect, FIFO underrun, etc.).

- **Increment diagnostic counters.** The HSM should maintain the diagnostic counters in the statistics table for every detectable error condition. The HSM should also increment the generic statistic *TotalTxMiscCount* if a fatal transmit error occurred that is not counted in any other standard counter. The fatal transmit error may be counted by the TSM using a media specific counter as well.

- **Attempt to send the packet again.** In the event the HSM has reached the maximum retry limit for sending a packet, discard the packet, increment *MSMTxFreeCount*, and transmit the next packet if one is waiting to be sent.

Using Shared Interrupts

An HSM can support shared interrupts provided that they are also supported by the host bus and the adapters which will share the interrupt. Interrupts can be shared if the bus is operating in level-triggered mode or external logic exists on the adapters sharing the interrupt.

- The MCA bus always uses level-triggered interrupts and can support shared interrupts.

- The PC/AT bus normally uses edge-triggered interrupts and will not support shared interrupts unless external logic exists on the adapters sharing the interrupt.

- The EISA bus normally uses edge-triggered interrupts, but each interrupt can be individually set to level-triggered mode in order to support shared interrupts.

A *DriverISR* routine which supports shared interrupts is very similar to one which does not. If the HSM supports shared interrupts, the ISR must perform the following operations:

- Immediately determine if the interrupt request is from its adapter. If not, return at once to the operating system ISR with *EAX* equal to a non-zero value and the zero flag cleared.

  ```
  or    al,01h   ; clear the zero flag
  ret            ; return to operating system ISR code
  ```

- If the interrupt request is from the HSM's adapter, the interrupt service routine should proceed. Upon completion, the ISR should return with *EAX* equal to zero and with the zero flag set.

  ```
  xor   eax,eax  ; zero eax & set the zero flag
  ret            ; returns to operating system ISR code
  ```

The HSM must indicate that the adapters are sharing interrupts by setting bit 5 in the *MLIDSharingFlags* field of the configuration table. The HSM must also initialize the *DriverParameterBlock* variable, *DriverEndOfChainFlag*, as described in the following table.

If the HSM:	**The HSM must:**	***DriverEndofChainFlag* value:**	
Supports shared interrupts	Set the *IOSharingInterrupt0Bit* (bit 5) in the *MLIDSharingFlags* field of the HSM's configuration table.	Zero	The shared interrupt vector is placed first on the shared interrupt chain. If another interrupt vector is requested after the original vector is placed at the head of the chain, the latter vector will be serviced first.)
		Non-zero	The shared interrupt vector is placed at the end of the shared interrupt chain by the operating system.
Does not support shared interrupts	Clear the *IOSharingInterrupt0Bit* (bit 5) in the *MLIDSharingFlags* field of the HSM's configuration table.	Not used.	

DriverISR Pseudocode

On Entry

EBP	Pointer to the Adapter Data Space
Dir Flag	is cleared
Interrupts	are disabled (Novell recommends interrupts remain disabled during the *DriverISR*)

On Return

Dir Flag	must be cleared
Interrupts	must be disabled
Note	no registers are preserved

```
DriverISR   proc

    The interrupt controller is normally serviced here. However, if we implement the
    DriverEnableInterrupt and DriverDisableInterrupt routines as recommended,
    the MSM and TSM will call these routines at the appropriate time.


;;;   MSMDisableHardwareInterrupt      ; Do NOT use for Multi-OS support
                                       ;  (see DriverEnableInterrupt)
;;;   MSMDoEndOfInterrupt              ; Do NOT use for Multi-OS support



CheckStatus:

    Get the controller's interrupt status


ReceiveEvent:
          .
          .
          .
    (check for receive errors and handle)
          .
          .
          .

 *** Setup a LookAhead Buffer ***

      mov       ecx, [ebp].MSMMaxFrameHeaderSize
      lea       edi, [ebp].LookAheadBuffer
      rep       insb
 *** Obtain an RCB for the Received Packet ***

       lea          esi, [ebp].LookAheadBuffer
       mov          ecx, HardwareReportedPacketSize
       call         <TSM>GetRCB

       if RCB is NOT available
            skip this packet
            jmp  CheckStatus

 *** Copy data and deliver RCB ***

       copy the packet data into the RCB fragment buffers
       call         <TSM>RcvComplete
       jmp          CheckStatus


TransmitEvent:
            .
            .
            .
     (check for transmit errors and handle/retry)
            .
            .
            .

TransmitComplete:

       reset retry counter to Maximum value
       mov          [ebp].TxInProgress, FALSE
       inc          [ebp].MSMTxFreeCount

 *** Transmit Next Packet In the Send Queue ***

       test         [ebp].MSMStatusFlags,TXQUEUED
       jz           Exit
       call         <TSM>GetNextSend
       jnz          Exit
       call         DriverSend


Exit:

;;;   MSMEnableHardwareInterrupt        ; Do NOT use for Multi-OS support
                                        ; (see DriverEnableInterrupt)
     MSMServiceEventsAndRet


DriverISR   endp
```

DriverPoll

The *DriverPoll* procedure is used if the HSM requires a poll-driven board service routine. This routine will typically perform functions similar to those of the *DriverISR* procedure.

Note: *DriverPoll* is normally not used by an interrupt-driven HSM, however, there may be some cases where polling is required or where polling is used in addition to the ISR.

To register the polling procedure, place a pointer to the procedure in the *DriverPollPtr* field of the *DriverParameterBlock*. The driver can then enable polling during initialization by calling *MSMEnablePolling*.

On Entry

EBP	Pointer to the Adapter Data Space
EBX	Pointer to the Frame Data Space
Interrupts	are disabled

On Return

EBP	must be preserved

Packet Transmission

This section provides a brief overview of the methods commonly used for packet transmission.

When sending a packet, a protocol stack assembles a list of fragment pointers in a transmit ECB and passes it to the LSL. The ECB is then transferred to the TSM where the information is processed and a TCB is constructed. The TCB structure consists of the assembled packet header and data fragment information. The TSM directs the TCB to the *DriverSend* routine which collects the header and packet fragments and transmits the packet.

Transmission Methods

The method of packet transmission selected is typically dependent on the adapter's data transfer method. The examples on the following pages are intended to illustrate a general flow of events. Refer to the appropriate MSM and TSM support call descriptions for detailed information.

In general, packet transmission involves the following steps:

- During *DriverInit*, initialize *MSMTxFreeCount* to the number of adapter transmit resources available.
- The TSM builds a TCB, checks to see if the driver can handle another transmit and if so, decrements *MSMTxFreeCount* and calls *DriverSend* (otherwise the TSM queues the packet).
- *DriverSend* will typically copy the media header and data fragments to the transmit buffer and start the transmission.
- The driver returns the TCB back to the TSM using *<TSM>Send-Complete*. This can be performed before the actual transmission is complete as long as all information has been collected from the TCB and the TCB is no longer needed (a "lying" send). The underlying transmit ECB will be placed in the LSL's holding queue until the HSM issues a service events command.
- Use the *MSMServiceEvents* macro to allow the LSL to call the transmit ECB's event service routine.
- When the actual transmission is complete, increment *MSMTxFree-Count*. This is typically performed during *DriverISR* after a transmit complete interrupt.

Programmed I/O, Shared RAM, and Host DMA

The sequence of events for transmitting a packet using programmed I/O, shared RAM, or host DMA adapters is described below.

HSM	1. Sets **MSMTxFreeCount** to the maximum number of packets that the adapter can buffer. (performed in **DriverInit**)

TSM	2. If the Ethernet TSM is used, ECX is set to the padded length of the packet. (This is the value that the adapter will send onto the wire, regardless of the value in the **TCBDataLen** field. In fact, the value in ECX is not equal to **TCBDataLen** if the packet is Ethernet 802.3 or Ethernet II and was evenized or if the packet was padded to 60 bytes.)
	3. Decrements **MSMTxFreeCount** and calls **DriverSend** with ESI pointing to a filled in TCB structure.
HSM	4. Calls **<TSM>SendComplete** or **<TSM>FastSendComplete** either after the packet has been buffered onto the adapter or after the transmission has been completed.
	5. Increments **MSMTxFreeCount** after the adapter completes the transmission (typically performed in **DriverISR**).

Bus Master

Option 1. This option is identical to the method described on the previous page for programmed I/O, shared RAM, and host DMA adapters.

Option 2. This method is recommended if the adapter is ECB aware and has sufficient adapter processor speed. It dramatically decreases the load on the server by reducing the host's process time.

HSM	1. Sets **DriverSendWantsECBs** to a non-zero value and sets **MSMTxFreeCount** to the number of packets that the adapter can process at one time. (performed in **DriverInit**)

TSM	2. Decrements **MSMTxFreeCount** and calls **DriverSend** with a pointer to the Frame Data Space in EBX and a pointer to the ECB in ESI.
HSM	3. Calls either **<TSM>SendComplete** or **<TSM>FastSendComplete** after the packet has been buffered onto the adapter or after the transmission has been completed.
	4. Increments **MSMTxFreeCount** after the adapter completes the transmission (typically performed in **DriverISR**).

DriverSend

The TSM calls *DriverSend* to transmit a frame onto the medium. *DriverSend* is provided a pointer to a Transmit Control Block (TCB). Refer to Chapter 3 for information on TCBs.

The HSM can assume that the TCB is valid for its LAN medium; it should not do consistency checking on the TCB fields. The HSM should also assume that it has the resources necessary to handle the transmit operation; it does not need to check to see if it has a transmit hardware resource available. The TSM performs flow control for the HSM. The TSM determines if the HSM can handle the packet by checking the value of *MSMTxFreeCount*.

On Entry

EBP	Pointer to the Adapter Data Space
EBX	Pointer to the Frame Data Space
ESI	Pointer to a Transmit Control Block (TCB) **
ECX	Padded length of the packet (Ethernet only)
Interrupts	are disabled. Novell recommends that system interrupts remain disabled during *DriverSend*.

On Return

Interrupts	must be disabled

****Note:** The *DriverSend* routine may request ECBs instead of TCBs by initializing the *DriverParameterBlock* variable *DriverSendWantsECBs* to a non-zero value (see Chapter 3). If *DriverSend* uses ECBs for packet transmission, it is responsible for building the proper media header (refer to Chapter 4 for additional information on ECB aware adapters). If the HSM uses ECBs instead of TCBs, it must **not** modify the transmit ECB's *BLink* field.

Pseudocode

```
Copy the MediaHeader from the TCB into a transmit buffer.
Copy the fragmented data from the TCB's fragment structure into a transmit buffer.
Give the command to send the packet.
Restore ESI to point to the beginning of the TCB

IF called from DriverISR
        Call <TSM>SendComplete                                    ("lying" send)
ELSE
        Call <TSM>FastSendComplete
ENDIF

Return
```

Multi-Operating System Support

Novell has been working towards a driver specification that allows v4.x HSMs to be transported to other 32-bit Intel-based operating system platforms without any code modification. In order to accomplish this task, two new driver routines have been added.

If you can control interrupts at the adapter, you should implement the *DriverEnableInterrupt* and *DriverDisableInterrupt* routines. Drivers that control interrupts at the adapter can be transported to other OS platforms where access to the PIC is restricted.

Previously drivers routines used the *MSMEnableHardwareInterrupt* and *MSMDisableHardwareInterrupt* macros to control the Programmable Interrupt Controller (PIC). You should only use these macros if interrupts can not be enabled and disabled at the adapter hardware level.

Note: In addition, if the HSM provides these two new calls, it must not use the *MSMDoEndOfInterrupt* macro or directly EOI the PIC. The MSM and TSM will perform all necessary operations at the appropriate time depending on the platform that the driver is running under.

The new specification also recommends avoiding the use of the *CLI* and *STI* instruction in HSM code.

Drivers should allow the MSM and TSM to control the interrupt state via calls to the *DriverEnableInterrupt* and *DriverDisableInterrupt* routines at the appropriate times. Novell recommends that the interrupt states remain unaltered during driver procedures. If a driver procedure must alter the interrupt state, it must restore it to the original state before returning.

DriverEnableInterrupt

This procedure enables interrupts at the adapter hardware.

On Entry

EBP	Pointer to the Adapter Data Space

On Return

EBP	must be preserved

```
DriverEnableInterrupt  proc

        Enable the adapter to generate interrupts
        ret

DriverEnableInterrupt  endp
```

DriverDisableInterrupt

This procedure disables interrupts at the adapter hardware.

On Entry

EBP	Pointer to the Adapter Data Space

On Return

EAX	If Interrupts are Non-Shareable EAX is zero If Interrupts are Shareable EAX is zero if the interrupt was generated by our board. EAX is non-zero if the interrupt was not ours. In this case, the TSM calls DriverEnableInterrupt on return from this routine.
EBP	must be preserved

```
DriverDisableInterrupt   proc

        Disable the adapter from generating interrupts
        mov   eax, <Appropriate Status>
        ret

DriverDisableInterrupt   endp
```

Control Procedures

The ODI specification requires drivers to implement the I/O control functions (IOCTLs) listed in the table below. The MSM and TSM development tools perform several of the required IOCTL functions without assistance from the HSM, as indicated in the table. The support modules will also "front end" all control functions and preserve any required registers. The HSM is responsible for implementing the control functions described in this section.

DriverReset and *DriverShutdown* are mandatory and must be present for the driver to function properly. The HSM should also provide the *DriverMulticastChange* and *DriverPromiscuousChange* procedures when the hardware supports these functions.

The *DriverStatisticsChange* and *DriverRxLookAheadChange* procedures are optional. These procedures allow drivers for intelligent adapters to update the statistics table or the LookAhead size only as needed. Refer to the *DriverParameterBlock* field descriptions in Chapter 3 for additional information on these two control procedures.

Drivers that support the Hub Management Interface must implement the *DriverManagement* procedure to handle management requests and commands as described in Chapter 8.

Control Function		Code Path
0	Get Configuration Table	MSM
1	Get Statistics Table	MSM -> DriverStatisticsChange
2	Add Multicast Address	MSM -> TSM -> DriverMulticastChange
3	Delete Multicast Address	MSM -> TSM -> DriverMulticastChange
4	Reserved	MSM
5	Shutdown Driver	MSM -> TSM -> DriverShutdown
6	Reset Driver	MSM -> TSM -> DriverReset
7	Reserved	MSM
8	Reserved	MSM
9	Set receive LookAhead size	MSM -> TSM -> DriverRxLookAheadChange
10	En/Dis Promiscuous Mode	MSM -> TSM -> DriverPromiscuousChange
11	En/Dis Receive Monitor	MSM -> TSM
12	Reserved	MSM
13	Reserved	MSM
14	Driver Management	MSM -> DriverManagement

DriverReset

DriverReset resets and initializes the adapter hardware. The routine may also test the hardware to verify that it is functional. If the driver has been temporarily shutdown, an application may call this routine to bring the board back into full operation.

When a reset is required, the TSM waits for transmissions in progress to complete and calls *DriverReset*.

From within the HSM, *DriverReset* may be called by *DriverInit*. It may also be called by *DriverCallBack* or *DriverISR* if the adapter had problems.

If the MSM calls *DriverReset*, and it returns successfully, the MSM resets the *MSMTxFreeCount* variable to the initial value set by the driver during initialization. If the MSM calls *DriverReset*, and the adapter cannot be reset, the MSM automatically calls *DriverShutdown*.

On Entry

EBP	Pointer to the Adapter Data Space
EBX	Pointer to the Frame Data Space
Interrupts	are disabled but may be enabled during the call

On Return

EAX	Zero if successful; otherwise EAX will return a pointer to a null-terminated error message.
Interrupts	are disabled

Pseudocode

```
Increment the reset statistics counter
Reset the hardware  (includes performing any hardware testing)
Call <TSM>UpdateMulticast
Set EAX to zero if successful
Return
```

DriverShutdown

DriverShutdown must place the hardware into a safe, inactive state. If the adapter is to be shut down permanently (indicated by the value in ECX), the MSM disables the adapter's interrupt immediately after this routine returns. As far as the HSM is concerned the only difference between a partial and a complete shutdown is the return of allocated memory.

Partial Shutdown - When a partial shutdown is required, the MSM sets *MSMStatusFlag,* waits for transmissions in progress to complete and returns the transmit ECBs. The MSM also sets bit 0 of the SharingFlags in the configuration table. *DriverReset* should be able to bring the adapter back into full operation.

Complete Shutdown - A zero value in ECX indicates a complete shutdown. As with a partial shutdown the MSM has set the flags, emptied the send queue, and also will return all resources not allocated directly by the HSM. If the HSM allocated memory using *MSMAlloc*, it must be returned using *MSMFree* before disabling the hardware.

The MSM automatically calls *DriverShutdown* when the *DriverReset* routine fails to reset the hardware. *MSMReturnDriverResources* and *MSMExitToDOS* also call *DriverShutdown*.

On Entry

EBP	Pointer to the Adapter Data Space
EBX	Pointer to the Frame Data Space
ECX	Zero if a permanent shutdown, otherwise a partial shutdown is required.
Interrupts	are disabled but may be enabled during the call

On Return

EAX	Zero if successful; FFFFFF84h on failure
Interrupts	are disabled

Pseudocode

```
IF a permanent shutdown
        return any memory using MSMFree
ENDIF

return any preallocated RCBs or queued TCBs
Disable Hardware
Set EAX = 0
Return
```

DriverMulticastChange

DriverMulticastChange updates the adapter to reflect the changes in the TSM's multicast address table. Novell recommends that all HSMs support multicast addressing if the hardware allows it. The following flags and variables must be initialized properly for the adapter's multicast mode.

- Bit 3 of the *MLIDModeFlags* is used to indicate whether or not multicast addressing is supported.
- Bits 9 and 10 of the *MLIDFlags* must be set appropriately to reflect the multicast mechanism or format used by the adapter/driver.
- The *DriverParameterBlock* variable, *DriverMaxMulticast*, must be set to reflect the maximum number of multicast addresses the adapter can handle.

The TSM maintains an internal table of multicast addresses. The TSM modules handle the addition and deletion of addresses in this table. Whenever the table changes, the TSM calls *DriverMulticastChange* to update the adapter's multicast filtering. The adapter may maintain its own multicast address table or use a hash table to filter incoming packets.

Adapter Multicast Filtering

The most common method used by adapters to filter incoming packets is hashing. When this is the adapter's method, *DriverMulticastChange* must recalculate and update the adapter's hash table. Hashing does not guarantee 100% multicast filtering; therefore, the TSM would look up incoming packets in its multicast address table to ensure that the packet's destination address is enabled.

In the case that the adapter keeps its own list of multicast addresses, this routine should cycle through the entries in the TSM's multicast address table and output each entry to the physical card. The TSM verifies that all addresses it places in its table are valid multicast addresses so the HSM does not need to validate them.

In either case, the HSM routine must read the TSM's multicast address table. Each entry in the table is 8 bytes long. The first 6 bytes are the address, and the last word is a use flag maintained by the TSM. If the use flag is non-zero, the entry contains a valid address.

```
MulticastEntryStruc  db 6 dup (?)   ; multicast addresses
MulticastInUse       dw 0           ; Non-zero if in use
```

The default method (if bits 9 and 10 of the *MLIDFlags* are zero) for handling multicast operations is as follows:

Ethernet and FDDI
On entry to this routine, ECX contains the number of valid entries in the multicast table. All valid entries will be contiguous, so the HSM does not necessarily need to check the *MulticastInUse* flag. If ECX is zero, multicast reception is disabled.

Token-Ring and PCN2
The TSM passes the 32-bit functional address in EDX. In this case ECX and ESI are normally not used.

Note: If an adapter is capable of supporting both group and functional addresses (and sets bits 9 and 10 in the *MLIDFlags* field of the configuration table accordingly), the *DriverMulticastChange* routine will receive both functional addresses and multicast table information .

RX-Net
DriverMulticastChange cannot be supported by RX-Net drivers.

On Entry

EBP	Pointer to the Adapter Data Space
EBX	Pointer to the Frame Data Space
ESI	Pointer to the Multicast Table (default for Ethernet or FDDI)
ECX	Number of valid entries in the Multicast Table (default for Ethernet or FDDI)
EDX	32-bit functional address (default for Token-Ring or PCN2)
Interrupts	are disabled on entry, but may be enabled during the routine

On Return

Note	EBX and EBP must be preserved
Interrupts	must be disabled on return

Pseudocode

```
Clear the hardware registers that filter incoming packets for multicast addresses
Get current multicast addresses from TSM's multicast table
Reload hardware register with new multicast address filtering values
Return
```

DriverPromiscuousChange

Adapters/drivers that can pass all packets to a monitor function in the protocol stack are said to have a promiscuous reception mode. *DriverPromiscuousChange* provides a means for the stack monitor function to enable or disable promiscuous reception.

Note: Realize that enabling promiscuous mode will have a detrimental impact on system performance.

A monitoring function examines packets sent from or received by an adapter. If promiscuous mode is supported, the monitoring function can request that the adapter enter promiscuous mode. When promiscuous mode is enabled, the driver should allow all packets (including bad packets if possible) to be passed up to the monitor function. Only one monitor function at a time may be registered with a driver.

Be aware that a monitor function may set the configuration table's *MLIDLookAheadSize* to a value other than the 18 byte default. This will in turn change *MSMMaxFrameHeaderSize*.

The *<TSM>GetRCB* and *<TSM>ProcessGetRCB* require the driver to indicate the status of the packet in EAX. EAX will always equal zero for Token-Ring, RX-Net, and FDDI. For Ethernet the status options are as follows:

- `EAX = zero for good packets`
- `EAX = non-zero for bad packets`

```
EAX bits are set as follows for bad packets:

 Bit 0  - CRC Error
 Bit 1  - CRC/Alignment Error
 Bit 2  - Runt packet (set by the Ethernet TSM)
 Bit 8  - Receive too big for ECB (set by the TSM)
 Bit 9  - No board number registered (set by the TSM)
 Bit 10 - Malformed packet (set by the TSM)
 Bit 31 - Driver shutting down (set by the TSM)
```

If the HSM does not support promiscuous mode, bit 13 of the *MLIDModeFlags* in the configuration table must be cleared and the *DriverPromiscuousChangePtr* field in the *DriverParameterBlock* must be zero.

Special Instructions **PCN2**

Drivers written with the PCN2L TSM must only use the *<TSM>GetRCB* and *<TSM>RcvComplete* combination. The driver must pass the status of the received packet to the *<TSM>RcvComplete* routine rather than the *<TSM>GetRCB*. PCN2 status bits are the same as those described above for Ethernet drivers.

On Entry

EBP	Pointer to the Adapter Data Space
EBX	Pointer to the Frame Data Space
ECX	Zero to disable promiscuous mode Non-zero to enable promiscuous mode For Token-Ring and FDDI the bits are defined as follows if ECX is non-zero: Bit 0 is set if MAC frames are to be received Bit 1 is set if non-MAC frames are to be received Both bits are set if all frames are to be received
Interrupts	are disabled but may be enabled during the call

On Return

Note	EBP and EBX must be preserved
Interrupts	are disabled

Pseudocode

```
IF requested to enable promiscuous mode
        send enabling command to hardware
ELSE
        send disabling command to hardware
```

DriverStatisticsChange (optional)

The *DriverStatisticsChange* routine allows the MSM's control procedure handler to notify drivers whenever an application requests IOCTL 1 (get driver statistics). This allows HSMs for intelligent adapters that maintain statistical information on board to update the statistics table in the Adapter Data Space only as needed (before the MSM passes it up to the requesting application).

For additional information, refer to the *DriverStatisticsChangePtr* field of the *DriverParameterBlock* in Chapter 3.

On Entry

EBP	Pointer to the Adapter Data Space
EBX	Pointer to the Frame Data Space
Interrupts	are disabled but may be enabled during the routine

On Return

Interrupts	are disabled
Note	EBP must be preserved

Pseudocode

```
Transfer statistics maintained by the hardware
  to the statistics table in the Adapter Data Space.
```

DriverRxLookAheadChange (optional)

The *DriverRxLookAheadChange* routine allows the MSM to notify drivers after an application invokes IOCTL 9 to set the LookAhead size. This IOCTL changes the *MSMMaxFrameHeaderSize* variable and the *MLIDLookAheadSize* field in the configuration table. Drivers can use this routine to inform intelligent adapters only when the size changes rather than constantly checking the value.

For additional information, refer to the *DriverRxLookAheadChangePtr* field of the *DriverParameterBlock*, the *MLIDLookAheadSize* in the configuration table, the *MSMMaxFrameHeaderSize* variable, and the *<TSM>GetRCB* procedure.

On Entry

EBP	Pointer to the Adapter Data Space
EBX	Pointer to the Frame Data Space
Interrupts	are disabled but may be enabled during the routine

On Return

Interrupts	are disabled
Note	EBP must be preserved

Pseudocode

```
Inform Adapter of new MSMMaxFrameHeaderSize (or new MLIDLookAheadSize)
```

DriverManagement (optional)

If a driver accepts management commands from outside NLMs (such as HMI or CSL), the MSM will call the *DriverManagement* routine to process the management requests.

Refer to Chapter 8 for a Hub management implementation of this procedure. See also the *DriverManagementPtr* field of the *DriverParameterBlock* in Chapter 3.

On Entry

EBP	Pointer to the Adapter Data Space
EBX	Pointer to the Frame Data Space
ESI	Pointer to the management ECB containing the request (see Chapter 8)
Interrupts	are disabled but may be enabled during the routine

On Return

Interrupts	are disabled
EAX	00000000h = Success; command ECB relinquished 00000001h = Success; command ECB queued FFFFFF88h = no such handle - ProtocolId not supported

Pseudocode

```
(refer to Chapter 8 for an example DriverManagement routine)
```

Timeout Detection

The *DriverAES* and *DriverCallBack* routines may be used when the HSM needs to be called at periodic intervals. Typically one of these routines is used to determine if a board has failed to complete a packet transmission, but other timed functions may be set up using these routines as well.

Which routine is used for the HSM's timeout handling depends on execution time constraints. While in *DriverCallBack,* the HSM may only use operating system routines that can be called at interrupt time. If any routines are used that must be called during process time only, then *DriverAES* should be used.

Note: RX-Net normally uses a specific routine, *DriverTxTimeout*, to handle transmit timeouts. This routine is required only when the RX-Net module is used. RX-Net drivers may also use the other two timing event routines.

DriverTxTimeout (RX-Net)

The RX-Net TSM calls *DriverTxTimeout* whenever a transmit has a software timeout. Under normal conditions, the HSM issues the **Disable Transmitter** command to the card. If the hardware does not require any special attention, the HSM simply returns.

DriverTxTimeout is called at interrupt time and should be optimized to be as efficient as possible. This procedure **must** be included when the HSM uses the RX-Net support module.

DriverAES / DriverCallBack

DriverAES is enabled (typically during initialization) by calling *MSMScheduleAESCallBack*, and *DriverCallBack* is enabled by calling the function *MSMScheduleIntTimeCallBack*. The use of these two MSM calls is explained in Chapter 7, but briefly, the MSM routines expect EAX to contain the desired time interval in ticks (1 tick ≈ 1/18 second).

Once enabled, the MSM invokes the routine automatically at the end of each interval with EBX pointing to the Frame Data Space and EBP pointing to the Adapter Data Space. Interrupts are enabled when *DriverAES* is called and are disabled on calls to *DriverCallBack*.

The actual content of the routines is entirely up to the developer. The pseudocode here illustrates using *DriverCallBack* to identify a send timeout error.

On Entry

EBP	Pointer to the Adapter Data Space
EBX	Pointer to the Frame Data Space
Interrupts	are enabled for *DriverAES* are disabled for *DriverCallBack*

On Return

Note	EBP must be preserved

Pseudocode

```
IF Transmit is in Progress

        IF Elapsed Transmit Time > Maximum Time for Transmit
                Increment appropriate error counter
                Reset the adapter
                Reset [ebp].MSMTxFreeCount

                Call <TSM>GetNextSend                          (check the send queue)
                IF TCB was available
                        Call DriverSend
                ENDIF
        ENDIF
ENDIF

Return
```

Removal

The NetWare operating system calls the driver's exit procedure, *DriverRemove*, when it receives the command to unload the driver. This procedure is described below.

DriverRemove

The *DriverRemove* procedure is called whenever the HSM is unloaded. The HSM's linker definition file must include the **"exit"** keyword followed by *DriverRemove*. Because this routine is called by the operating system, it must preserve the C registers EBP, EBX, ESI and EDI.

This routine must set EAX to the value of the *DriverModuleHandle* from the *DriverParameterBlock* and call *MSMDriverRemove*. The MSM handles MLID deregistration, returns all driver resources, and calls *DriverShutdown* before returning.

On Entry

Interrupts	can be in any state

On Return

Interrupts	are preserved

Pseudocode

```
DriverRemove  proc

      push      ebx, ebp, esi, edi
      mov       eax, DriverModuleHandle
      call      MSMDriverRemove
      pop       edi, esi, ebp, ebx
      ret

DriverRemove  endp
```