4

MSL Driver Procedures

DriverInitialize

The MSL driver must provide an initialization procedure that performs the tasks involved in hardware registration, initialization, and testing. The operating system calls the *DriverInitialize* routine each time a *load* command is issued for the driver.

Drivers are typically written so that a *load* command must be issued for each host adapter. In a future release of NetWare SFT III, loading multiple MSL adapters will be supported. (See the *Dual Mirrored Server Links* description in Chapter 1.) Drivers should also allow the operator to load the driver with a single specified adapter, to selectively enable only the desired host adapters.

DriverInitialize must determine and reserve hardware configuration options. It also exchanges any required information with NetWare and brings the adapter up to operational mode. If *DriverInitialize* fails to initialize the adapter, it returns an error status to the operating system and the driver's code is unloaded.

The *DriverInitialize* procedure performs the following tasks:

- Allocate resource tags
- Determine the hardware configuration
- Register the hardware configuration options
- Set hardware interrupts
- Initialize and test adapter hardware
- Register the driver with the OS
- Schedule callback events for error detection and recovery
- Report and recover from any initialization errors
  If initialization fails:
  -Release hardware options
  -Release interrupts
  -Cancel scheduled callback events
  -Return allocated memory
- Return initialization status to the caller (OS)

The remainder of this section describes the *DriverInitialize* tasks in detail. An example of an MSL driver's initialization procedure can be found in the MSL driver listing in Appendix E.

Allocate Resource Tags

Resource tags are used by the operating system to identify and control various hardware and system resources. Drivers are required to allocate several types of resource tags before making certain system calls. The resource tags are then validated by the OS when the calls are made and are used to track the requested resource. If a module fails to free up allocated resources prior to termination then the OS can perform the cleanup operations so the resources are not lost to the system.

Normally the driver acquires all needed resource tags before performing any other driver initialization functions. The resource tags are not deallocated by the driver or returned to the operating system, since the OS routines accomplish this automatically upon module termination. (See the *AllocateResourceTag* description in Chapter 5 for details.)

Determine Hardware Options

The driver must determine the hardware configuration information needed for the *IOConfigurationStructure*. This includes options 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 as described in Appendix C.

The driver uses the *ParseDriverParameters* procedure to obtain and validate hardware configuration options entered on the load command line and to query the operator for any required parameters which were not specified. The *ParseDriverParameters* procedure requires an *AdapterOptionStructure* 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 selected values are used to fill in the adapter's *IOConfigurationStructure*. (See the *ParseDriver-Parameters* description in Chapter 5 for details.)

Register Hardware Options

When all needed information has been determined for the driver's *IOConfigurationStructure*, the *DriverInitialize* routine must register the hardware options with the operating system. The OS is informed of the configuration using the *RegisterHardwareOptions* procedure. This routine reserves the hardware configuration for the adapter and will notify the driver of any conflicts with existing hardware in the system.

Set Hardware Interrupts

Driver initialization routines must allocate requested interrupts by calling *SetHardwareInterrupt*.

Interrupts can be shareable or non-shareable. The driver indicates that it can share the interrupt by setting the appropriate bit in the *CFlags* field of the *IOConfigurationStructure* and by setting the *ShareFlag* parameter passed to the *SetHardwareInterrupt* routine. If operating in shared mode, the driver's ISR must provide logic for handling shared interrupts. It must determine if an interrupt is for an adapter associated with the driver and return this indication back to the OS.

For further information see the *SetHardwareInterrupt* description in Chapter 5 and the *DriverISR* section later in this chapter.

Initialize the Host Adapters

The driver can only initialize the host adapter and register it with the OS *after* the necessary hardware options have been validated and reserved. The driver must not issue instructions to I/O ports, access shared RAM, etc..., until hardware registration is completed, unless they are standard MCA or EISA system ports used to determine the slot configuration.

The procedure for initializing an adapter depends entirely on the requirements for the particular hardware design. NetWare places no specific requirements on adapter initialization, except that when *DriverInitialize* returns, the host adapter should be fully initialized and ready for operation. It may even require procedures such as loading host adapter firmware.

Verify Host Adapter Operability

The driver should test the host adapter during initialization to ensure that it is operational. If the host adapter exhibits a problem or in any way fails testing, it should not be registered with the OS. A brief message describing the problem should be displayed for the operator's benefit and the driver should proceed with the error recovery steps outlined later in this chapter.

Load Coprocessor Firmware

NetWare custom data can be anything that might be required by a driver. For example, the driver may need to read in firmware to be loaded into a co-processor board. To define the custom data file, use the CUSTOM keyword in the driver's linker definition file followed by the filename (custom data files are simply appended to the driver module). NetWare passes the custom data file's handle, starting offset, size, and the *ReadRoutine* address to the initialization procedure, where it must be saved upon entry if custom data is going to be read by the driver. The initialization procedure can read the file into memory by calling the *ReadRoutine* using the syntax shown below:

```
ReadRoutine  (
      LONG  CustomDataFileHandle,
      LONG *CustomDataOffset,
      LONG *CustomDataDestination,
      LONG  CustomDataSize );
```

The driver must supply the destination in memory according to the needs of the host adapter. *Some adapters only support word or doubleword moves to or from shared RAM*, and will not support moves with other widths or alignments.

The *ReadRoutine* does byte moves to the supplied destination logical address. The driver may need to allocate a block of memory to read the custom data into prior to moving it to the destination shared RAM in the adapter using word or doubleword moves. The *ReadRoutine* returns an error code if the driver attempts to read beyond the end of the custom data.

The custom data file is not interpreted in any way by NetWare, and may be in any form. The custom data file is typically raw machine code that can be downloaded to a coprocessor card, and may be prepared in any way desired, using any language processors or linkers desired.

Register the Driver

The driver must register with the operating system by calling the NetWare routine *RegisterServerCommDriver* (described in detail in Chapter 5). Four entry points into the MSL driver are passed with this call:

- *DriverSend*
- *DriverBuildSend*
- *DriverEmergencySend*
- *DriverControl*

In addition, an *MSLResourceTag* is required which allows the OS to track all of the MSL-requested OS resources.

Schedule Callbacks

Drivers use the *ScheduleNoSleepAESProcessEvent* routine to schedule callbacks to the *DriverTimeOut* procedure. This driver procedure is used to detect and recover from timeout conditions. After the card is operational, the callback procedure monitors the adapter's performance. If a significant delay occurs in the adapter's operation, the procedure may intervene and cause a retry or notify the OS of the error.

An *AESEventStructure* and *AESProcess* resource tag are required to schedule the driver callback. (See the *DriverTimeOut* description later in this chapter for more information.)

Return Initialization Status

A return status of zero in EAX indicates a successful initialization. If the driver returns a non-zero status (indicating an error), the driver is removed from server memory. This allows a module's initialization routine to prevent the OS from using the driver.

Error Reporting and Recovery

Reporting Errors - Errors that occur during initialize can be reported at the console using the support routine *OutputToScreen* or optionally *QueueSystemAlert* (see Chapter 5). The driver is passed a ScreenHandle when NetWare calls the *DriverInitialize* routine. The handle should be saved by the driver for use *only* during the driver initialization routine.

Note: The OS routine *QueueSystemAlert* can be called at any time from any level of execution, including from an ISR, and does not require a Screen Handle.

Recovering from Errors - If an error occurs during initialization, all hardware resources allocated from the server must be returned using *DeRegisterHardwareOptions*. The driver must also free allocated interrupts by calling *ClearHardwareInterrupt* and return any memory that has been allocated. Any scheduled AES timers must also be canceled by calling *CancelNoSleepAESProcessEvent*. In addition, the adapter should be disabled so that it cannot interfere with the operation of the server.

DriverInitialize Summary

1 Save any parameters passed on the stack from the OS that are needed for later use.

2 Allocate all resource tags required by the driver.

3 Verify that the operator is not attempting to load more adapters than the driver will support.

4 (Optional) Call *GetHardwareBusType* to determine the bus type of the processor. This is used by drivers which can support cards on multiple bus types. It may also be desirable to verify that the bus type of the server is compatible with the driver being loaded.

5 (Optional) Allocate any required memory (if not statically defined in the driver data segment) using *AllocSemiPermMemory*.

6 Determine the Hardware Configuration

   (a) For MCA or EISA machines, search the slots for the adapter ID and build a slot list for the *AdapterOptionStructure*.

   (b) Get the hardware configuration information or slot to use by calling *ParseDriverParameters* with the appropriate *NeedBitMap* value and *AdapterOptionStructure*. This call fills in the *IOConfigurationStructure* with the selected values.

   (c) For MCA or EISA adapters the slot field in the *IOConfiguration-Structure* now contains the appropriate slot number which can be used to determine the configuration information by...

      - reading the POS registers for MCA
      - reading the configuration block for EISA

7 Call *RegisterHardwareOptions* to reserve the options and check for any hardware conflicts.

8 Register the driver's ISR by calling *SetHardwareInterrupt*.

9 Initialize and test the adapter hardware.

10 (Optional) Read firmware or other custom data from the Custom Data File and load into the coprocessor. Start the coprocessor executing the loaded firmware.

11 Register the driver with the OS using *RegisterServerCommDriver*.

12 Initialize two global variables:

- *MaxCommDriverDateLength*
- *PacketSizeDriverCanHandle*

14 Schedule driver callback timers (for driver timeout recovery) using *ScheduleNoSleepAESProcessEvent* or *ScheduleSleepAES-ProcessEvent*.

15 Return initialization status to the operating system.

Error Steps

A Cancel any active driver callback AES timers using either *CancelNoSleepAESProcessEvent* or *CancelSleepAESProcessEvent*, depending on the AES timer type.

B Unhook the driver's ISR by calling *ClearHardwareInterrupt*.

C Release registered hardware resources to the OS by calling *DeRegisterHardwareOptions*.

D Return any dynamically allocated memory back to OS by calling *FreeSemiPermMemory*.

E Return to the OS with an error status (EAX = non-zero value).

Example

```
;*****************************************************************************
;* DriverInitialize                                                          *
;*****************************************************************************
;*                                                                           *
;* Stack Parameters:                                                         *
;*                                                                           *
;*     Parm0 = ModuleHandle                  Parm5 = LoadableModuleFileHandle  *
;*     Parm1 = ScreenHandle                  Parm6 = ReadRoutine             *
;*     Parm2 = CommandLine                   Parm7 = CustomDataOffset        *
;*     Parm3 = (reserved)                    Parm8 = CustomDataSize          *
;*     Parm4 = (reserved)                                                    *
;*                                                                           *
;*****************************************************************************

Align 16
DriverInitialize        proc

        CPush
        mov     ebp, esp
        pushfd
        cli

        ;*************************************************************************
        ;* Allocate all resource tags used by MSL                                *
        ;*************************************************************************

        push    MSLSignature
        push    OFFSET RTagMessage_MSL
        push    [ebp + Parm0]
        call    AllocateResourceTag
        add     esp, 3 * 4
        or      eax, eax
        mov     MSLDriverResourceTag, eax
        mov     ebx, OFFSET ErrorGettingRTag_MSL
        jz      DisplayMessageExit

        push    IORegistrationSignature
        push    OFFSET RTagMessage_IORegistration
        push    [ebp + Parm0]
        call    AllocateResourceTag
        add     esp, 3 * 4
        or      eax, eax
        mov     DriverConfiguration.CIOResourceTag, eax
        mov     ebx, OFFSET ErrorGettingRTag_IORegistration
        jz      DisplayMessageExit

        push    InterruptSignature
        push    OFFSET RTagMessage_Interrupt
        push    [ebp + Parm0]
        call    AllocateResourceTag
        add     esp, 3 * 4
        or      eax, eax
        mov     InterruptResourceTag, eax
        mov     ebx, OFFSET ErrorGettingRTag_Interrupt
        jz      DisplayMessageExit

        push    TimerSignature
        push    OFFSET RTagMessage_Timer
        push    [ebp + Parm0]
        call    AllocateResourceTag
        add     esp, 3 * 4
        or      eax, eax
        mov     IntHoldOffEvent.TResourceTag, eax
        mov     ebx, OFFSET ErrorGettingRTag_Timer
        jz      DisplayMessageExit
```

```
        push    AESProcessSignature
        push    OFFSET RTagMessage_AESProcess
        push    [ebp + Parm0]
        call    AllocateResourceTag
        add     esp, 3 * 4
        or      eax, eax
        mov     TimeOutEvent.AESRTag, eax
        mov     HoldOffEvent.AESRTag, eax
        mov     ebx, OFFSET ErrorGettingRTag_AESProcess
        jz      DisplayMessageExit


        ;*************************************************************************
        ;* Parse which port and interrupt to use                                 *
        ;*************************************************************************

        push    [ebp + Parm1]
        push    [ebp + Parm2]
        push    NeedsIOPort0Bit OR NeedsInterrupt0Bit
        push    0
        push    0
        push    OFFSET AdapterOptions
        push    0
        push    OFFSET DriverConfiguration
        call    ParseDriverParameters
        add     esp, 8 * 4
        or      eax, eax
        mov     ebx, OFFSET ErrorParsingIOMessage
        jnz     DisplayMessageExit


        ;*************************************************************************
        ;* Register Hardware Options                                             *
        ;*************************************************************************

        push    0
        push    OFFSET DriverConfiguration
        call    RegisterHardwareOptions
        add     esp, 2 * 4
        or      eax, eax
        mov     ebx, OFFSET ConflictingHardwareMessage
        jnz     DisplayMessageExit


        ;*************************************************************************
        ;* Set Interrupt Vector                                                  *
        ;*************************************************************************

        push    OFFSET ExtraEOIFlag
        push    CHAIN_SET_REAL_MODE
        push    0
        push    InterruptResourceTag
        push    OFFSET DriverISR
        movzx   eax, BYTE PTR DriverConfiguration.CInterrupt0
        push    eax
        call    SetHardwareInterrupt
        add     esp, 6 * 4
        or      eax, eax
        mov     ebx, OFFSET ErrorGettingInterruptMessage
        jnz     DeRegisterHardware


        ;*************************************************************************
        ;* Initialize and Test the MSL Adapter                                   *
        ;*************************************************************************

        call    HardwareInit                    ;returns ptr to error message
        jnz     DriverInitHardwareError
        mov     FirstTimeInit, 0                ;disable testing the adapter
                                                ;hardware again
```

```
        ;************************************************************************
        ;* Register the MSL driver with the OS                                  *
        ;************************************************************************

        push    OFFSET DriverControl
        push    OFFSET DriverEmergencySend
        push    OFFSET DriverBuildSend
        push    OFFSET DriverSend
        push    OFFSET DriverConfiguration
        push    MSLDriverResourceTag
        call    RegisterServerCommDriver
        add     esp, 6 * 4
        or      eax, eax
        mov     ebx, OFFSET ErrorRegisteringMSLMessage
        jnz     ErrorRegisteringDriver

        mov     MaximumCommDriverDataLength, MAX_PACKET_SIZE
        mov     PacketSizeDriverCanNowHandle, MAX_PACKET_SIZE

        ;************************************************************************
        ;* Start Timeout Callbacks                                              *
        ;************************************************************************

        push    OFFSET TimeOutEvent
        call    ScheduleNoSleepAESProcessEvent
        add     esp, 1 * 4




;*****************************************************************************
;* DriverInitialize Successful Exit                                          *
;*****************************************************************************

        popfd
        xor     eax, eax                                ; Return Success!
        CPop
        ret



;*****************************************************************************
;* DriverInitialize Error Paths                                              *
;*****************************************************************************

ErrorRegisteringDriver:
DriverInitHardwareError:

        ;************************************************************************
        ;* Unhook from Interrupt vector                                         *
        ;************************************************************************

        push    OFFSET DriverISR
        movzx   eax, BYTE PTR DriverConfiguration.CInterrupt0
        push    eax
        call    ClearHardwareInterrupt
        add     esp, 2 * 4

        ;************************************************************************
        ;* Deregister hardware options from OS                                  *
        ;************************************************************************

DeRegisterHardware:

        push    OFFSET DriverConfiguration
        call    DeRegisterHardwareOptions
        add     esp, 1 * 4
```

```
        ;*************************************************************************
        ;* Display Error Message in EBX                                          *
        ;*************************************************************************

DisplayMessageExit:

        push    ebx                                     ; Pointer to error string
        push    [ebp + Parm1]                           ; Screen Handle
        call    OutputToScreen                          ; Display Error Message
        add     esp, 2 * 4

        popfd
        or      eax, -1                                 ; Return Failure
        CPop
        ret


DriverInitialize        endp









;*****************************************************************************
;* HardwareInit                                                              *
;*****************************************************************************

HardwareInit    proc


        ; (Adapter-specific code to bring up adapter to operational mode)


HardwareInitSuccess:

        xor     eax, eax
        ret


HardwareInitError:

        mov     ebx, OFFSET HardwareInitErrorMessage
        or      eax, -1
        ret


HardwareInit    endp
```

DriverControl

[Non-Blocking]

Syntax

long DriverControl (Parm0, Parm1 , , ,) ;

Parameters

Parm0 = Function_Number
Specifies which control function is being requested.
0 = GetMSLConfiguration
1 = GetMSLStatistics

Parm1 = Buffer_Pointer
Pointer to the buffer to copy Configuration or Statistics.
If pointer=0, return size of Configuration or Statistics.

(All other parameters are function-dependent)

Return Value

EAX contains a completion code.

```
EAX = 0 (Success)

EAX = Size
```
If *Parm1* is zero, return the size of the structure or table.

```
EAX = 0FFFFFF81h (BAD_COMMAND)
```
Bad *Function_Number* was passed

Requirements

Called at process level.

Description

The MSL *DriverControl* interface routine is the entry point for all the driver's control subroutines. The driver must implement the *GetMSLConfiguration* and *GetMSLStatistics* procedures, both detailed in this section. These procedures provide statistical and configuration information to the caller.

GetMSLConfiguration provides the original caller with a copy of the MSL configuration structure in the buffer specified by *Buffer_Pointer*. If the pointer is zero, just return the *size* of the configuration structure in EAX.

GetMSLStatistics provides the original caller with a copy of the MSL statistics table in the buffer specified by *Buffer_Pointer*. If the pointer is zero, just return the *size* of the statistics table in EAX.

Example

```
Align 16

;*************************************
;* Control Procedure Vector Information *
;*************************************

ControlProcedures      dd    GetMSLConfiguration
                       dd    GetMSLStatistics
ControlProceduresEnd   equ   $

MaxControlNumber       equ   ((ControlProceduresEnd-ControlProcedures)/4)-1




;*****************************************************************************
;*                              DriverControl                                *
;*****************************************************************************
;*                                                                           *
;*  Function 0 = GetMSLConfiguration                                         *
;*  Function 1 = GetMSLStatistics                                            *
;*                                                                           *
;*  Stack Parameters:                                                        *
;*     Parm0 = Function Number                                               *
;*     Parm1 = Pointer to the buffer to copy Configuration or Statistics.    *
;*             If pointer=0,  return size of Configuration or Statistics.    *
;*                                                                           *
;*****************************************************************************

Align 16
DriverControl   proc


   CPush                                ;save C registers
   mov   ebp, esp                       ;get stack base
   pushfd                               ;save flag state
   cli                                  ;clear interrupts

   mov   ebx, [ebp + Parm0]             ;get requested function #
   cmp   ebx, MaxControlNumber          ;check if request is valid
   ja    InvalidControlProcedure        ;jump if not

   call  ControlProcedures [ebx * 4]    ;table thru routine


DriverControlExit:

   popfd                                ;restore flags
   CPop                                 ;restore registers
   ret


InvalidControlProcedure:

   mov   eax, BAD_COMMAND               ;flag invalid status
   jmp   DriverControlExit


DriverControl   endp
```

Example (continued)

```
;*****************************************************************************
;*                GetMSLConfiguration - Driver control procedure 0           *
;*****************************************************************************

Align 16
GetMSLConfiguration   proc


   mov   edi, [ebp + Parm1]                ;get buffer pointer
   or    edi, edi                          ;get size only? (edi=0)
   jz    SHORT GetMSLConfigurationSize     ;jump if so

   mov   ecx, DriverConfigurationSize      ;copy the configuration
   mov   esi, OFFSET DriverConfiguration
rep movsb

   xor   eax, eax
   ret


GetMSLConfigurationSize:

   mov   eax, DriverConfigurationSize      ;get configuration size
   ret


GetMSLConfiguration   endp







;*****************************************************************************
;*                 GetMSLStatistics - Driver control procedure 1             *
;*****************************************************************************

Align 16
GetMSLStatistics   proc


   mov   edi, [ebp + Parm1]                ;get buffer pointer
   or    edi, edi                          ;get size only? (edi=0)
   jz    SHORT GetMSLStatisticsSize        ;jump if so

   mov   ecx, DriverStatisticsSize         ;copy the statistics
   mov   esi, OFFSET DriverStatistics
rep movsb

   xor   eax, eax
   ret


GetMSLStatisticsSize:

   mov   eax, DriverStatisticsSize         ;get statistics size
   ret


GetMSLStatistics   endp
```

DriverSend

[Non-blocking]

On Entry

The following registers contain the message header parameters to be sent to the other server.

EAX	Parameter
EBX	Parameter
ECX	Parameter (length of message data, may be zero)
EDX	Parameter
ESI	Parameter (address of message data)
EDI	Parameter

On Return

EAX must be set to zero if successful. A non-zero value indicates failure and the driver must call *ServerCommDriverError*.

All registers may be modified upon return except segment registers, the stack pointer, and EAX (which must have the completion code).

Requirements

The OS will call this routine at either Interrupt or Process level. Interrupts will be disabled and are *required* to remain disabled.

Description

This procedure is called by the SFT III operating system to send a single message to the other server. On entry to the *DriverSend* routine, the registers EAX, EBX, ECX, EDX, ESI, and EDI contain the parameter values for the message header to be sent to the other server. These 6 registers will be used by the *receive* routine in the other server when calling *ReceiveServerCommPointer*.

The driver must copy the message header and message data (if any) to the adapter and initiate the packet's transmission. ECX is the length of the message data in bytes and ESI contains the address of the data (if the length is non-zero).

The driver completely controls the server's ability to call the *DriverSend* and *DriverBuildSend* procedures by controlling the value of the global variable *PacketSizeDriverCanNowHandle* as follows:

`negative` =	*DriverSend* cannot be called by the OS
`zero` =	*DriverSend* can send a message header only (no data)
`> zero` =	*DriverSend* can send a message header plus data up to the number of bytes indicated by *PacketSizeDriverCanNowHandle*.

The OS may call *DriverSend* any time the value in *PacketSizeDriver-CanNowHandle* will accommodate a message the OS has queued to send. The operating system will attempt to send more than one message without waiting for an acknowledgment unless prevented by

the driver. The driver prevents this by placing a negative value in *PacketSizeDriverCanNowHandle*.

Note: The driver should *not* attempt to use the send procedure's execution time to receive a packet. It should simply validate the packet, place the packet data into its transmit buffer, initiate the transmission sequence, and return.

Example

```
;*****************************************************************************
;* DriverSend                                                                *
;*****************************************************************************
;*                                                                           *
;*      On Entry:                                   On Exit:                 *
;*                                                                           *
;*      EAX = OS parameter                          EAX = Not saved          *
;*      EBX = OS parameter                          EBX = Not saved          *
;*      ECX = OS parameter/Length of Message Data   ECX = Not saved          *
;*      EDX = OS parameter                          EDX = Not saved          *
;*      EBP = Not Defined                           EBP = Not saved          *
;*      ESI = OS parameter/Pointer to Message Data  ESI = Not saved          *
;*      EDI = OS parameter                          EDI = Not saved          *
;*                                                                           *
;*      Interrupts Disabled                                                  *
;*                                                                           *
;*****************************************************************************

Align 16
DriverSend      proc


        mov     PacketSizeDriverCanNowHandle, -1        ;inform OS we're busy
        mov     TxPacketMessageCount, 1                 ;sending one message
        inc     TransmitMsgCount                        ;update statistics counter

        ;*************************************************************************
        ;* Build Message Packet in Transmit Buffer                               *
        ;*************************************************************************

        ; Note:  this code assumes the hardware can accept loading of message data
        ; even if the channel is busy with another transmit


        (Setup packet header and message header in transmit buffer here)


        or      ecx, ecx                                ;any data with message?
        jz      DriverSendReady                         ;skip data copy if not


        (Copy message data to transmit buffer here:  ecx=size  esi=addr)



DriverSendReady:

        call    TransmitMessagePacket
        inc     TransmitPacketCount
        xor     eax, eax
        ret


DriverSend      endp
```

Example (continued)

```
;*****************************************************************************
;* TransmitMessagePacket                                                     *
;*****************************************************************************

Align 16
TransmitMessagePacket   proc

        cmp     TransmitInProgress, TRUE        ;if a transmit is in progress...
        je      PutMessageTransmitOnHold        ;...jump (can't send msg now)


        (Initiate the transmit of the loaded message packet here)


        ;*************************************************************************
        ;* Begin watching for Adapter Timeout Errors                             *
        ;*************************************************************************

        mov     TransmitInProgress, TRUE
        mov     TimeOutEvent.AdapterTimeOutTime, ADAPTER_TIMEOUT_COUNT

        ;*************************************************************************
        ;* Begin watching for Message Timeout Errors                             *
        ;*************************************************************************

        mov     MessageInProgress, TRUE
        mov     eax, ServerCommACKTimeOut
        mov     TimeoutEvent.MessageTimeoutTime, ax

        mov     MessageTransmitPending, FALSE
        ret


PutMessageTransmitOnHold:

        mov     MessageTransmitPending, TRUE
        ret


TransmitMessagePacket   endp
```

DriverSend Summary

Transmit a single message

1 Stop OS from sending anything
   (*PacketSizeDriverCanNowHandle* = -1)
   (Not mandatory if dual port transmit buffer)
2 *TxPacketMessageCount* ← 1
3 Increment statistics counter (*TransmitMsgCount*)
4 Build media-required header in transmit buffer
5 Build message header (6 registers, 24 bytes)
6 Check ECX for data to copy?
   -yes: copy data to adapter
   -no: skip
7 Check if channel is available
   -yes: transmit packet
   -no: set *MessageTransmitPending* flag and exit
8 Start transmit timeout sequence as soon as transmit is sent
9 Return status to OS

DriverBuildSend

[Non-blocking]

On Entry

The following registers contain the message header parameters to be sent to the other server.

EAX	Parameter
EBX	Parameter
ECX	Parameter (length of message data, may be zero)
EDX	Parameter
ESI	Parameter (address of message data)
EDI	Parameter

On Return

EAX must be set to zero if successful. A non-zero value indicates failure and the driver must call *ServerCommDriverError*.

All registers may be modified upon return except segment registers, the stack pointer, and EAX (which must have the completion code).

Requirements

This procedure is called at interrupt level. Interrupts will be disabled and are *required* to remain disabled. The driver must *not* use this procedure's execution time to receive a packet.

Description

The *DriverBuildSend* procedure is used to build multi-message packets. The operating system queues messages when the driver is busy transmitting another message. After the *DriverISR* receives a message acknowledgement and notifies the OS (by calling *SendServerCommCompletedPointer*), it must obtain any queued messages by calling *GetNextPacketPointer*.

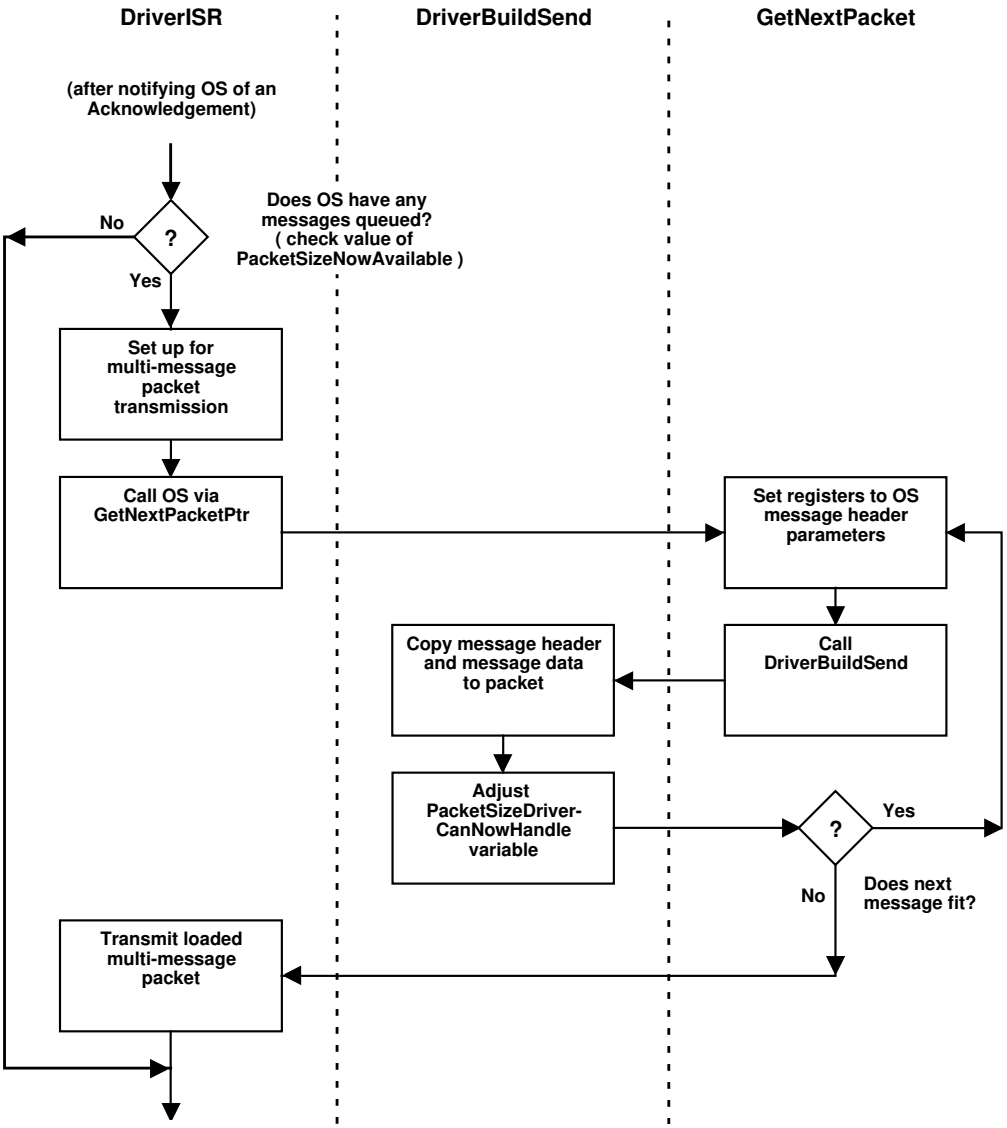
If there are messages queued, calling this procedure initiates a possible multimessage building sequence. During *GetNextPacket* the *DriverBuildSend* procedure is called repeatedly to build the multimessage packet. *GetNextPacket* will stop calling the *DriverBuildSend* routine only when the driver indicates, through the value in *PacketSizeDriverCanNowHandle*, that it has no more room for additional messages or when the OS has no more messages to send.

The *DriverBuildSend* procedure must copy the message header and message data (if any) to the transmit buffer. On entry, the registers EAX, EBX, ECX, EDX, ESI, and EDI contain the parameter values for the message header. The message data length (in bytes, not including the header) is in ECX, and the message data address (assuming ECX is not zero) is in ESI. *DriverBuildSend* must then update the *PacketSizeDriverCanNowHandle* variable and return.

MSL drivers should place a limit on the maximum number of messages transmitted in a multi-message packet. Because the driver operates with interrupts disabled during this procedure, as well as during message processing, allowing an unlimited number of messages may prevent other critical processes from executing in a timely manner. This can cause significant degradation in operating system performance and may result in mirrored server failures.

Note: As a starting point, we recommend using a maximum of 128 messages per packet and optimizing for the particular MSL design. The optimum value will vary from driver to driver.

The *DriverBuildSend* procedure and its interaction with the *DriverISR* and *GetNextPacket* code is illustrated in the flow diagram below.



Example

```
;*****************************************************************************
;* DriverBuildSend                                                           *
;*****************************************************************************
;*                                                                           *
;*      On Entry:                                   On Exit:                 *
;*                                                                           *
;*      EAX = OS parameter                          EAX = Not saved          *
;*      EBX = OS parameter                          EBX = Not saved          *
;*      ECX = OS parameter/Length of Message Data   ECX = Not saved          *
;*      EDX = OS parameter                          EDX = Not saved          *
;*      EBP = Not Defined                           EBP = Not saved          *
;*      ESI = OS parameter/Pointer to Message Data  ESI = Not saved          *
;*      EDI = OS parameter                          EDI = Not saved          *
;*                                                                           *
;*      Interrupts Disabled                                                  *
;*                                                                           *
;*****************************************************************************

Align 16
DriverBuildSend proc


        (Build Message Header)


        or      ecx, ecx                            ;any data with message
        jz      DriverBuildSendDone                 ;skip data copy if not


        (Copy Message Data to adapter:  ecx=size  esi=addr)


DriverBuildSendDone:

        inc     TransmitMsgCount                    ;update statistics counter
        inc     TxPacketMessageCount                ;updata message count

        cmp     TxPacketMessageCount, MAX_MESSAGE_COUNT ;max # of messages yet?
        je      HitMaxMessageCount                  ;jump if so

        sub     PacketSizeDriverCanNowHandle, MESSAGE_HEADER_SIZE
        sub     PacketSizeDriverCanNowHandle, ecx

        xor     eax, eax                            ;indicate success
        ret                                         ;return to OS

HitMaxMessageCount:

        mov     PacketSizeDriverCanNowHandle, -1    ;indicate no more msgs
                                                    ;in this packet
        xor     eax, eax
        ret                                         ;return to OS


DriverBuildSend endp
```

DriverEmergencySend

[Non-blocking]

Parameters

None

Return Values

None (This procedure must return; it will be returning to Abend)

Requirements

This procedure can be called at either Process or Interrupt level. Interrupts are disabled on entry and *must* remain disabled.

Description

The *DriverEmergencySend* procedure is called by the SFT III operating system to notify the other server that this server is no longer operational. It may be called at ANY time.

The *DriverEmergencySend* procedure should make its best attempt to send an emergency packet or signal informing the other server of the emergency. If necessary, the driver may loop with interrupts disabled waiting for an operation such as a transmit to complete in order to implement this function. However, care must be taken to avoid an endless loop. The driver can also abort a function to send the emergency signal.

Important !

If the driver is unable to successfully send the emergency packet, it must still return to the caller. Avoid performing any operations that could cause a machine lockup or prevent the return to the caller.

Some drivers may not be able to implement this feature, but it is essential to accelerating the fault tolerance handling and recovery of SFT III.

Receiving an Emergency Notification

When an MSL driver receives an *emergency notification* packet, it notifies the OS of the emergency by calling the *ServerCommDriverError* routine (described in Chapter 5).

If the driver is in a message holdoff state when an *emergency notification* packet is received, *ServerCommDriverError* must not be called until all heldoff messages that have already been acknowledged are delivered to the operating system. Otherwise, there is a potential for divergence of the two servers' states, which would destroy the mirror.

Example

```
;*****************************************************************************
;* DriverEmergencySend                                                       *
;*****************************************************************************

Align 16
DriverEmergencySend     proc


        cmp     TransmitInProgress, FALSE       ;if not transmitting now
        je      EmergencySendReady              ;fire Emergency packet

        mov     ecx, MAX_EMERGENCY_WAIT         ;set adapter specific wait
                                                ;(this is the maximum loop time
                                                ; required to complete a transmit)


EmergencySendWaitLoop:

        in      al, 61h                         ;else waste time
        in      al, 61h                         ;(same for all speed machines)


        (Read Adapter Status)
        (If transmit channel is now available....jmp  EmergencySendReady)

        loop    EmergencySendWaitLoop


ForceEmergencySend:

        (If possible, cancel the adapter's current transmission and force send)


EmergencySendReady:

        (Transmit the Emergency Notification)

        ret


DriverEmergencySend     endp
```

DriverEmergencySend Summary

The goal of the *Emergency Notification* is to notify the other server of this servers failure as soon as possible. If needed, the driver may delay, but it should make a best attempt to send the emergency packet. However, interrupts must remain disabled and the driver must not wait indefinitely.

1 If the transmit channel is available:

Send the *Emergency Notification* packet, then exit.

2 If the transmit channel is not available:

Delay up to the maximum wait time for the channel to be made available (*don't wait forever*).

If the channel is still not available, either try to force a send or exit. (Do not force the send if doing so could cause the machine or the bus to hang.)

DriverISR

The driver's Interrupt Service Routine, *DriverISR*, is called by the system ISR (which actually receives the interrupt) for all hardware interrupts. An interrupt may indicate one of the following events:

- Message packet received
- Acknowledgment received
- Holdoff notification received
- Emergency notification received
- Reception error encountered
- Transmission complete
- Transmission error encountered

Each of these cases is detailed on the following pages. (Some adapters may not generate an interrupt for each of the above events.) An example of an MSL driver's interrupt service routine can be found in the MSL driver listing in Appendix E.

The driver ISR must perform the following tasks:

- Clear the interrupt on the adapter
- Issue End of Interrupt commands (EOIs)
- Perform all functions to service the interrupt, such as:
  - Process messages
  - Cancel/start timer events
  - Retry unsuccessful operations
  - Post completion status
  - Check for additional operations to initiate
- Return (do not `iret`) to the caller (the system ISR)

All of the driver ISR code must run with interrupts disabled due to its function within the driver and SFT III OS architecture. *Under no circumstances* is a driver ISR allowed to make any calls to *Blocking* routines.

Receiving a Message Packet

Upon successful reception of a message, the MSL driver should perform the following functions:

1 Verify message packet reception
(Is it a valid message, no receive errors)
2 Send a message acknowledgment
3 Call *ReceiveServerCommPointer* with the message header
4 Examine the completion code returned to determine what action to take with the message.

ReceiveServerCommPointer is a global variable that contains a pointer to the operating system's current receive handler. The MSL driver should load EAX, EBX, ECX, EDX, EDI, and ESI with the six values passed as the message header from the other server and notify the OS of the message by calling the *ReceiveServerCommPointer* routine. This routine must be called for each message received from the other server. This call returns with one of five possible completion codes (CCode), listed below.

Note: The OS may modify the ECX and ESI registers (effectively bypassing or ignoring the data). The modified ECX and ESI registers must be used to copy the message if required. Since ECX may have changed, the adapter still needs to use the original length to find the end of this message or the beginning of the next message.

CCode = 0 (OK)

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

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

CCode = 1 (OK with Callback)

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

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

After copying the data, the driver should make a callback to the Receive handler. This is done by calling the address specified by EDX. Prior to making the callback, the registers must be restored to the original message header values with the exception of ECX and ESI (use the new possibly modified values).

CCode = 2 (Holdoff)

Signals the driver to place the message on hold for redelivery at a later time. The driver may either send a request asking the other server to resend the packet or save the packet and attempt to deliver the message at a later time. This completion code is used by the operating system to throttle the incoming packets.

Note: The operating system needs to run before it will be able to accept this packet. An immediate attempt to redeliver this packet without relinquishing control will be fruitless. Redelivery can be accomplished by setting up an AES and interrupt time callback event that relinquishes control, then triggers an attempt to redeliver the packet.

Care must be exercised to ensure that the packet is not delivered to the OS twice.

CCode = 3 (Holdoff)

Same as CCode 2 above.

CCode = 4 (Ignore)

The driver should ignore this message.

Once the driver has successfully delivered the packet data to the OS, it should send an acknowledgment to the sender. (The driver may send the acknowledgment before copying the data, but must then be able to receive an ensuing packet from the other server.)

Note: The OS may call the *DriverSend* procedure while executing the *ReceiveServerCommPointer* procedure (unless the driver has set *PacketSizeDriverCanNowHandle* to a negative value). The driver must be able to send a packet at this point. To prevent a possible "deadlock" situation, if *DriverSend* is called from within *ReceiveServer-CommPointer* and the receiving driver cannot receive system commands like the ACK while a message is being held off, it must guarantee the delivery of the ACK before the packet, just in case the packet does get held off. (A "deadlock" case is where both servers are waiting on something from the other server that cannot be delivered because of the Holdoff state.)

Important: A return code of 2 or 3 requires the MSL driver to enter a receive hold state. The receive hold state requires the MSL driver to handle errors differently. Normally when the MSL driver is NOT in the receive hold state, it calls the OS immediately (via *ServerCommDriverError*) upon detection of any errors. However, when the MSL driver is in a receive hold state, it must not call the OS with notification of an error. The driver must instead note the error, finish delivering all received messages to the OS, and only then notify the OS of the detected error. The driver must notify the OS of detected errors in this manner to preserve the mirrored state of the servers.

Receiving an Acknowledgement

When an acknowledgement is received for a message previously sent to the other server, the MSL driver should notify the operating system by making a call via the *ServerCommCompletedPointer*. The number of messages being acknowledged must be passed in EBP to the OS procedure *ServerCommCompletedPointer*.

Note: The call via the *ServerCommCompletedPointer* allows the operating system to move ahead to the next event or state. Because the acknowledgment permits the operating system to move ahead, *it is critical that the receiver only send an acknowledgment when it has successfully received the packet*. Otherwise, the servers may get out of sync causing potential data loss in the event of a system failure.

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

The MSL driver must transmit additional messages the OS may have queued up and waiting. The process of obtaining and sending queued messages is covered in the *DriverBuildSend* routine description.

Receiving an Emergency Notification

An emergency packet (or signal) is used to tell the operating system that the other server has failed. If an emergency packet is received, the driver should perform the following functions:

1 Increment Diagnostic Counter: The driver should maintain a count of the number of emergency packets (signals) it has received

2 Check if the driver is in a message holdoff state. If an *emergency* packet is received while the driver is in a holdoff state, the OS must not be called until the holdoff condition is over. This applies only if the ACK for the held off message has already been sent. (Otherwise there is a potential for the two servers' states to diverge, and the mirror would be lost.)

   Note: If the held off message is part of a multi-message packet, the remaining messages in the packet must also be delivered before the OS is notified of the emergency.

3 Call *ServerCommDriverError*

Although the other server is non-functional, this server is still operational. Therefore, the driver should still continue to function as normal. In other words, after making the call to *ServerComm-DriverError*, the driver should continue to send and receive packets as if nothing had happened.

Handling Receive Errors

If a receive error is encountered, the MSL driver should perform the following actions:

1 Increment Diagnostic Counters: The driver should maintain diagnostic counters for every detectable error condition on the adapter. Although some adapters provide greater diagnostic support than others, the driver should attempt to pinpoint the specific cause of the error.

2 Attempt to Recover: An attempt to recover from a *Receive* error (such as requesting the sender to retransmit the packet or resetting the adapter), should be careful not to interfere with the normal operation of the adapter. For example, the error handler should not interfere with a transmit that may be in progress.

   Care must be taken to ensure that recovery attempts do NOT cause the same packet to be delivered multiple times.

3 Signal an Unrecoverable Error: If the driver detects a failure that causes communication to cease, it should notify the operating system by making a call to *ServerCommDriverError*.

The receive routine is responsible for validating each message received, and for delivering the message data to the operating system. As with any communications driver, the integrity of the data is of utmost importance. The MSL communications adapter and associated MSL driver must ensure the validity of the data delivered to the operating system. This may include attempts to correct or retransmit packets.

The driver must ensure that attempts at recovery do not compromise the integrity of the system. If a retry attempt could result in even a small possibility that erroneous messages could be delivered to the operating system (this may cause an abend on the secondary server), the driver should report that the link is no longer valid rather than retry the operation.

Transmit Complete

Each time a packet is successfully transmitted by the adapter, the driver should increment the *TransmitPacketCount* statistics counter and cancel any adapter watchdog timers.

If a transmission error is encountered perform the steps outlined below.

Transmit Errors

Note: If the MSL driver is in a message holdoff state, do not notify the OS of the transmit error until no longer in the holdoff state.

If a transmit error is encountered, the MSL driver should perform the following actions:

1 Increment Diagnostic Counters: The driver should maintain diagnostic counters for every detectable error condition on the adapter. Although some adapters provide greater diagnostic support than others, the driver should attempt to pinpoint the specific cause of the error.

2 Attempt to Recover: An attempt to recover from a Transmit error (such as attempting to retransmit the packet or resetting the adapter) should be careful not to interfere with the normal operation of the adapter. For example, the Transmit error handler should not interfere with a packet reception that may be in progress.

3 Signal an Unrecoverable Error: If the driver detects a failure that causes communication to cease, it should notify the operating system by making a call to *ServerCommDriverError* with an error code of zero, signaling a hardware failure.

The driver is responsible for delivering each message intact and in order to the other server. The integrity of the data propagated by the MSL driver is of utmost importance. The communications adapter hardware and associated driver must ensure the correctness of the data delivered to the other server. This may include attempts to retransmit messages; however, the driver must guarantee that attempts at recovery do not compromise the integrity of the system.

DriverHoldOff / DriverIntHoldOff

This section describes the MSL's Holdoff handling procedures used to redeliver messages to the OS.

The operating system will sometimes indicate to the MSL driver that it wants a message "held off" and redelivered at a later time. The OS passes this holdoff indicator to the MSL driver via a *ReceiveServer-CommPointer* completion code of 2 or 3.

The MSL driver, upon receiving the Holdoff indicator, must set up a redelivery system. This may necessitate sending a *HoldOff Notification* to the other server to prevent an inadvertent time-out due to the holdoff. (The *DriverTimeout Routine* is described in the next section.)

It is important to allow the server to execute briefly before trying to redeliver a message which has been given a holdoff status, so that the server can process and clear the cause for the holdoff. However, it is also very important to present the message to the server very soon after giving it time to process, so that the server throughput does not suffer appreciably.

In order to meet both requirements, the MSL driver should implement a dual callback mechanism to redeliver a message: One mechanism to redeliver as soon as possible, and another to guarantee redelivery of a message in a fixed amount of time. The guaranteed delivery mechanism is needed to prevent inadvertent timeouts.

An *AESSleep* timer should be used for the quick-retry requirement, because it will typically occur more quickly and frequently than any other callback mechanism offered by the OS.

An interrupt time (timer) callback should be used to guarantee that, even if the OS is very busy, a redelivery attempt will be made every timer tick. The interrupt time callback is also necessary to ensure that a *holdoff notification* can be sent to the other server, so that it does not timeout waiting for the acknowledgment (if, for example, a process does not relinquish control for several ticks).

Sample Holdoff routines are shown on the following pages. The *HoldOffDeliverMessageToOS* procedure called in this example (which performs the actual message redelivery), as well as the ISR routine (which scheduled the holdoff callbacks), are listed in the sample MSL Driver template in Appendix E.

Example

```
;*****************************************************************************
;* DriverHoldOff                                                             *
;*****************************************************************************

Align 16
DriverHoldOff   proc    near


        CPush
        cli

        cmp     HoldStateFlag, 0
        je      DriverHoldOffExit

        mov     BackOffAmount, 0


AttemptToRedeliverMessage:

        call    HoldOffDeliverMessageToOS
        cmp     HoldStateFlag, 0
        je      CancelDriverIntHoldOff

        inc     BackOffAmount
        mov     eax, BackOffAmount
        mov     HoldOffWaitLoopCount, eax


HoldOffWaitLoop:

        call    CRescheduleLast
        cmp     HoldStateFlag, 0
        je      DriverHoldOffExit

        dec     HoldOffWaitLoopCount
        jnz     HoldOffWaitLoop
        jmp     AttemptToRedeliverMessage



CancelDriverIntHoldOff:

        mov     edx, OFFSET IntHoldOffEvent
        call    CancelInterruptTimeCallBack


DriverHoldOffExit:

        CPop
        ret


DriverHoldOff   endp
```

Example

```
;*****************************************************************************
;* DriverIntHoldOff                                                          *
;*****************************************************************************

Align 16
DriverIntHoldOff        proc    near


        cmp     HoldStateFlag, 0                ;message still on hold?
        je      DriverIntHoldOffExit            ;if not, DriverHoldOff got
                                                ;   it delivered already

        call    HoldOffDeliverMessageToOS       ;else redeliver message now

        cmp     HoldStateFlag, 0                ;message still on hold?
        je      DriverIntHoldOffExit            ;if not, we got it delivered

        mov     edx, OFFSET IntHoldOffEvent     ;otherwise, reschedule callback
        call    ScheduleInterruptTimeCallBack   ;   to this routine

DriverIntHoldOffExit:

        ret                                     ;exit if done


DriverIntHoldOff        endp
```

DriverTimeout

The *DriverTimeOut* routine is a callback routine used by the MSL driver for monitoring all packet transmissions. If the other server has not acknowledged the reception of the last message packet in a timely manner, the MSL should notify the OS by calling the *ServerComm-DriverError*. If the MSL driver is in a message holdoff state, it should not take any action until it is out of the holdoff state.

The driver must not inadvertently time out messages which have been given a holdoff status: If holdoff could prevent acknowledgments to outstanding sends, care should be taken to avoid inadvertently timing out on the sends for that reason (the Holdoff state blocking the ACKs).

If the MSL driver is not in a message holdoff state, it may notify the OS of the break in the communication link by passing an error code to the OS procedure *ServerCommDriverError*.

The error code TIME_OUT_ERROR is used for message timeout errors and HARDWARE_ERROR for adapter timeout errors.

If a timeout error occurs, the driver should clear all receive variables, transmit variables, and system indicator variables, bringing the MSL driver into a state where it can initiate a new communication link. (Note: Do not clear the statistics counters.)

Example

```
;*****************************************************************************
;*  DriverTimeOut                                                            *
;*****************************************************************************
;*                                                                           *
;*  This routine will be executed every tick.  If on entry both the          *
;*  AdapterTimeOutTime and MessageTimeOutTime counters are zero, the         *
;*  adapter is idle.  This routine handles timeout events if the adapter     *
;*  did not complete the transmit or if an acknowledgement is not received.  *
;*                                                                           *
;*  Assumes:    Interrupts are enabled                                       *
;*                                                                           *
;*****************************************************************************

Align 16
DriverTimeOut   proc


        CPush
        cli



CheckIfAdapterTimedOut:

        cmp     TimeOutEvent.AdapterTimeOutTime, 0
        je      CheckIfMessageTimedOut

        dec     TimeOutEvent.AdapterTimeOutTime
        jnz     CheckIfMessageTimedOut

        ;*** Adapter Timed Out ***

        inc     AdapterTimedOutCount                    ;custom statistics counter
        mov     TransmitInProgress, FALSE
        mov     TimeOutEvent.AdapterTimeOutTime, 0

        mov     eax, HARDWARE_ERROR
        cmp     HoldStateFlag, 0
        je      NotifyError

        mov     HardwareErrorPending, TRUE
        jmp     DriverTimeOutExit



CheckIfMessageTimedOut:

        cmp     TimeOutEvent.MessageTimeOutTime, 0
        je      DriverTimeOutExit

        dec     TimeOutEvent.MessageTimeOutTime
        jnz     DriverTimeOutExit

        ;*** Message Timed Out ***

        inc     MessageTimedOutCount                    ;custom statistics counter
        mov     MessageInProgress, FALSE
        mov     TimeOutEvent.MessageTimeOutTime, 0

        mov     eax, TIME_OUT_ERROR
        cmp     HoldStateFlag, 0
        je      NotifyError

        mov     TimeOutErrorPending, TRUE
        jmp     DriverTimeOutExit
```

Example (continued)

```
NotifyError:

        push    eax
        call    ServerCommDriverError                   ;call OS
        add     esp , 1 * 4                             ;clean up stack
        inc     ServerCommErrorCount


DriverTimeOutExit:

        push    OFFSET TimeOutEvent
        call    ScheduleNoSleepAESProcessEvent
        add     esp, 1 * 4

        CPop
        ret


DriverTimeOut   endp
```

DriverRemove

NetWare calls the *DriverRemove* routine only once. This procedure returns all resources the driver has allocated from the server, after which it returns to the caller. The driver code is removed from server memory upon return.

NetWare requires the *DriverRemove* procedure to remove all related structures and finally the driver's code image from file server memory. The procedure is called by the console command *unload*. Please note that all structures for multiple adapter cards are unloaded by a single call to *DriverRemove*.

The *DriverRemove* procedure must perform the following steps:

1 Disable interrupts.

2 Disable the adapter.

3 If the driver is in HoldOff state, it must continue to redeliver acknowledged messages until HoldOff is finished.

4 Cancel any active callback events. The driver must deactivate any callback event timers by calling *CancelNoSleepAESProcessEvent*, *CancelSleepAESProcessEvent*, or *CancelInterruptTimeCallBack* depending on the timer type.

5 Deregister from the Mirrored Server Link (MSL) interface, by calling *DeRegisterServerCommDriver*.

6 Restore interrupts allocated by the driver by calling *ClearHardwareInterrupt*.

7 Release hardware resources by calling *DeRegisterHardwareOptions*.

8 Deallocate any memory allocated for the driver by calling *FreeSemiPermMemory* (for each block of memory obtained previously from the *AllocSemiPermMemory* routine).

9 Return.

Example

```
;*************************************************************************
;* DriverRemove                                                          *
;*************************************************************************

Align 16
DriverRemove    proc


        CPush
        pushfd
        cli


        ;*************************************************************************
        ;* Unhook from Interrupt vector                                          *
        ;*************************************************************************

        push    OFFSET DriverISR
        movzx   eax, BYTE PTR DriverConfiguration.CInterrupt0
        push    eax
        call    ClearHardwareInterrupt
        add     esp, 2 * 4


        ;*************************************************************************
        ;* See if we are currently in a Holdoff state                            *
        ;*************************************************************************

        cmp     HoldStateFlag, 0
        je      CancelCallBackEvents


        ;*************************************************************************
        ;* Wait until the holdoff state is finished to cancel                    *
        ;*************************************************************************

TryAgain:

        call    CRescheduleLast
        cmp     HoldStateFlag, 0
        jne     TryAgain


CancelCallBackEvents:

        mov     edx, OFFSET IntHoldOffEvent
        call    CancelInterruptTimeCallBack

        push    OFFSET HoldOffEvent
        call    CancelSleepAESProcessEvent
        add     esp, 1*4

        push    OFFSET TimeOutEvent
        call    CancelNoSleepAESProcessEvent
        add     esp, 1*4


        ;*************************************************************************
        ;* Deregister driver from OS                                             *
        ;*************************************************************************

        push    MSLDriverResourceTag                    ;pass Resource tag
        call    DeRegisterServerCommDriver              ;remove the driver.
        add     esp, 1*4
```

Example (continued)

```
        ;*************************************************************************
        ;* Deregister hardware options from OS                                   *
        ;*************************************************************************

        push    OFFSET DriverConfiguration
        call    DeRegisterHardwareOptions
        add     esp, 1*4

        popfd
        CPop
        ret


DriverRemove    endp
```