2

MSL Driver Overview

Introduction

The Mirrored Server Link requires a special driver customized to the specifications of the SFT III server-to-server communications interface. This chapter summarizes the basic functions of the MSL driver. Specific issues that influence the development of the MSL driver will also be addressed.

MSL Driver Functionality

The basic responsibility of the MSL driver is to transmit and receive message packets through the mirrored server communications link. It is crucial that this process be as efficient as the hardware will allow.

The MSL driver must guarantee message delivery and is responsible for message integrity. The driver must ensure that the data it receives is error-free before returning message acknowledgments. Although the operating system provides an optional high-level checksum of packets, the driver is ultimately responsible for packet transmission integrity.

The driver must provide duplicate packet suppression. The server checks for duplicate packets; failure to suppress duplicate packets will cause the secondary server to fail.

An MSL driver must detect Mirrored Server Link failures and notify the operating system accordingly. This includes detecting breakdowns in the communications link, as well as failures in the companion server. The driver uses operational timeouts and notifications from the hardware to detect these failures.

MSL Drivers are NLMs

Mirrored Server Link (MSL) drivers for NetWare SFT III are NetWare Loadable Modules (NLMs). MSL drivers must be converted from object form to a special Loadable Module form, referred to as a super-object file. The driver must provide basic NLM interface routines that adhere to NLM interface specifications and facilitate dynamic loading and unloading.

The driver must also follow prescribed procedures, and must utilize the defined standard routines outlined in Chapter 5 to interface to the operating system. The driver must export its required NLM interface routines and import any required NetWare SFT III operating system interface routines. The details of creating an MSL driver NLM are discussed in Appendix A.

MSL Driver Components

Driver Procedures

Every MSL driver must provide certain mandatory procedures in order to function properly. Additional procedures may be added to support the specific requirement of the MSL adapter.

The MSL driver consists of the following base procedures:

- *DriverInitialize*
- *DriverControl*
- *DriverSend*
- *DriverBuildSend*
- *DriverEmergencySend*
- *DriverISR*
- *DriverRemove*

Brief descriptions of the above MSL procedures are provided on the following pages. These descriptions are general and do not necessarily apply in every case. Chapter 4 provides detailed descriptions of the required procedures.

DriverInitialize

The *DriverInitialize* routine is called by the operating system when the driver is *loaded*. It performs all adapter hardware initialization and testing. The *DriverInitialize* routine also uses operating system support calls to perform the following tasks:

- Allocate required resource tags
- Determine the hardware configuration
- Register the hardware configuration with the OS
- Set up for the Interrupt Service Routine
- Register the driver with the OS
- Schedule timer events for error detection and recovery

DriverControl

The MSL *DriverControl* routine is the entry point for all the driver's control procedures. The driver must implement two control procedures, *GetMSLConfiguration* and the *GetMSLStatistics*. These procedures provide statistical and configuration information to the caller.

DriverSend

The operating system calls the MSL's *DriverSend* procedure to transmit a single message to the other server. The driver is responsible for building the message packet from information provided by the OS. The driver should not attempt to use the *DriverSend* procedure's execution time to receive a packet. It should simply build the message packet, initiate the transmission, and begin a transmit timeout sequence.

DriverBuildSend

The *DriverBuildSend* procedure allows the MSL driver to send multiple messages in a single packet. The operating system queues messages when the driver is busy transmitting another message. Once the driver receives a message acknowledgement, it must obtain any queued messages. *DriverBuildSend* is used to build the multi-message packet.

DriverEmergencySend

The SFT III operating system uses the *DriverEmergencySend* procedure to inform the other server that this server is about to go down. The *DriverEmergencySend* procedure is responsible for sending the emergency notification (or packet). The driver should make its best attempt to inform the other server of the emergency. This may even include aborting the current operation to send the emergency signal.

DriverISR

The Interrupt Service Routine (ISR) must handle all interrupts generated by the adapter. An interrupt may indicate any one of the following conditions:

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

DriverISR is called by the system ISR (which actually receives the interrupt) for all hardware interrupts. 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)

The DriverISR code must run with interrupts disabled due to its function within the driver and SFT III architecture.

DriverRemove

The *DriverRemove* routine is called by the console command processor in response to an *unload* command. This procedure must cancel all timers and operations, remove all requests, and free all resources allocated by the driver. On return from this procedure, the driver is removed from the file server memory.

Data Structures and Variables

In addition to the procedures described in the previous section, the driver must also contain certain data structures and variables. The primary structures include the *Driver Statistics Table* and the *Driver Configuration Structure*.

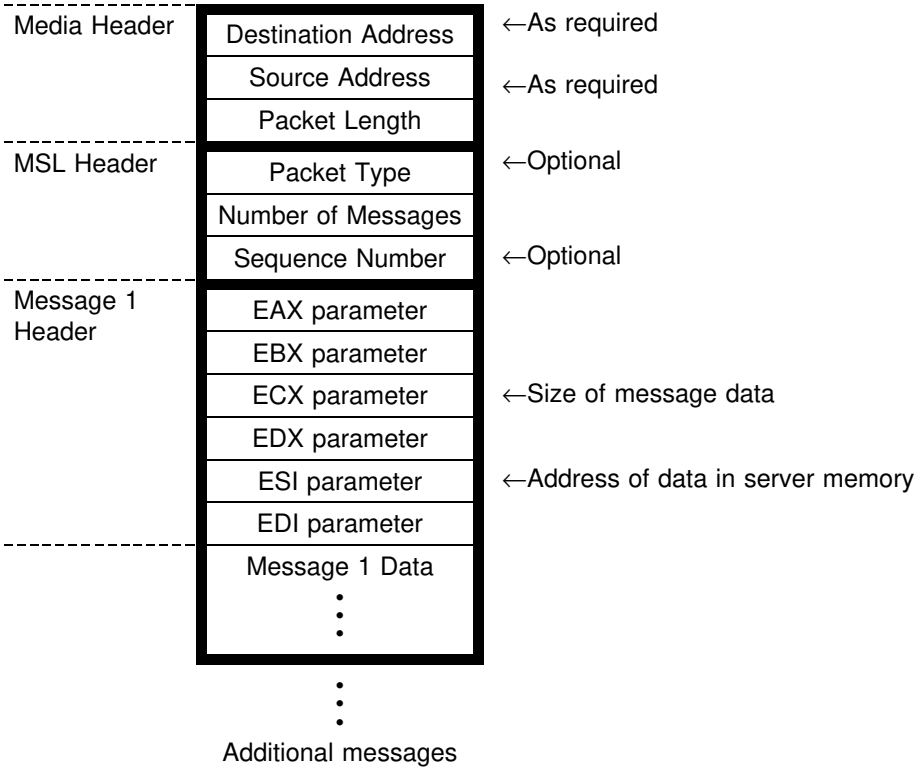
Chapter 3 provides detailed descriptions of all required MSL driver data structures and variables.

Message Packet Format

A *message packet* is composed of one or more messages preceded by a *media header* (as required) and an *MSL header* containing information about the packet. A *message* consists of a *message header* (parameters corresponding to registers EAX, EBX, ECX, EDX, ESI, and EDI), followed by variable-length *message data*. A message could also consist of the message header only (zero-length data).

A majority of the messages will be about 24 bytes in length. As stated above, a single message packet can contain one or more messages. Multi-message packets are used when the operating system has several messages queued up or back logged for transmission. Thus for the sake of efficiency, the capability to handle large packets containing many messages is desirable.

The following example is one possible layout for a message packet. The actual packet format as well as the elements in the header portions may vary according to the requirements of your particular MSL implementation.



MSL Driver Environment

MSL drivers operate as an integral part of the NetWare SFT III operating system. Therefore, the developer must understand the following operating system characteristics when writing the MSL driver code.

Multi-Tasking, Non-Preemption OS

The NetWare SFT III operating system is multi-tasking and non-preemptive. Non-preemptive means that a process does not lose its thread of execution unless it deliberately relinquishes control. A phrase frequently used to describe this mode of operation is "run to completion." Driver routines, therefore, must not dominate system resources. Processes must behave in a friendly fashion and periodically relinquish control so other processes will have an adequate opportunity to execute.

32-Bit Protected Mode

The NetWare operating system executes in 32-bit protected mode, therefore, *all drivers must run in 32-bit mode.* Some NetWare APIs allow drivers to call functions that switch to real mode, cause a real mode interrupt, then restore 32-bit mode operation. These APIs are for initialization and configuration only, and if used otherwise, would severely impact server performance.

Drivers may always assume SS=ES=DS, but should not assume that CS is identical to DS.

Reentrancy

NetWare SFT III currently does not support reentrancy for Mirrored Server Link drivers.

Execution Times

Drivers must comply with several execution level restrictions defined by the OS. The two principal execution times are process time and interrupt time. As you write the driver, you must be aware of which routines are called at process time, at interrupt time, or at either time. The times at which a driver procedure is called affect which operating system support routines the driver can access. The execution time restrictions for the OS calls are documented in Chapter 5. The execution levels and their associated environments are described in the following sections. Driver routines must not violate the defined environment of the execution level.

Process Time

At process level, the code executing is the process currently scheduled by the OS. Driver routines called at this level also execute as an extension of the current executing process. There are two types of process time routines: *Blocking* routines, which may suspend their own execution to allow other processes on the run queue to execute, and *Non-Blocking* routines, which do not suspend execution.

A **Blocking Process Level** routine may suspend its execution until a specified function completes, by making calls to system routines that suspend the process. All routines described in Chapter 5 may be called at this level, whether indicated as *blocking* or *non-blocking*.

Interrupts are normally enabled upon entry to routines at this level. Drivers may need to disable interrupts for a period of time during these process-level routines to call system support routines or to maintain driver integrity. Care should be taken to disable interrupts for the absolute minimum period required to accomplish the necessary tasks. *Disabling interrupts for any significant period will cause server performance degradation and poor response*.

Routines at this level may execute for a few milliseconds before returning to the operating system. If a routine requires more than a few milliseconds to complete, the driver must cause a task switch by calling *CRescheduleLast* or *DelayMyself*. This allows other NetWare processes to execute in a timely manner. Failure to do so may cause the operating system to indicate the violation on the server console.

The driver-defined procedures called at the *blocking process level* are:

- *DriverInitialize*
- *DriverRemove*
- *Driver SleepAESProcessEvent(s)*

A **Non-Blocking Process Level** routine may not suspend its execution. Routines called from this level may not make calls to blocking routines. Only system routines indicated as *non-blocking* may be called at this level.

Interrupts are disabled upon entry to routines at this level, with the exception of the *NoSleepAESProcessEvent* entry.

The driver-defined procedures called at the *non-blocking process level* are:

- *Driver NoSleepAESProcessEvent(s)*

Interrupt Time

Interrupt level routines execute under the identity of the current process (the process whose execution was interrupted). Because the current process is unknown upon entry at this level, *interrupt level routines cannot make blocking calls* or in any way affect a thread switch. Only system routines indicated as *non-blocking* may be called during interrupt level. Interrupts are always disabled upon entry to routines at this level.

Driver-defined procedures called at *interrupt level* are:

- *DriverISR*
- *Driver Interrupt Time Callback(s)*

The system ISR (which actually receives the interrupt) saves all registers, initializes the segment registers, and clears the direction flag before calling the driver's ISR. The driver must simply issue EOI commands, service the interrupt, and return (do not use `iret`).

MSL drivers must run with interrupts disabled during the entire ISR.

Process or Interrupt Time

Procedures that can be called at either process or interrupt time must not make any blocking calls or violate the rules related to execution at interrupt time.

Procedures which may be called at *either process or interrupt level* are:

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

C Calling Conventions

Most operating system support routines provided for drivers, as well as the several driver routines exported to the NetWare OS, must follow C subroutine calling conventions. The following sections detail the conventions to which drivers must adhere.

Being Called by a C Routine

Several driver routines called by NetWare must use C subroutine calling conventions. Any parameters passed to the driver routines are pushed on the stack in C-compatible reverse order. Driver routines that require use of passed parameters must retrieve them from the stack. The driver routines called must save registers EBP, EBX, ESI, and EDI on the stack upon entry and must restore them before returning to the caller.

Calling C Support Routines

The majority of the NetWare operating system support routines use standard C conventions. A few register-based routines are provided for the sake of efficiency. In most cases, parameters are passed on the stack when calling system routines. Drivers do not need to save registers EBP, EBX, ESI, and EDI when making system calls. These registers are saved by the routines called. All calls are NEAR due to the flat memory model used by NetWare.

To call a NetWare routine:

1) push all variables on the stack
2) call the routine
3) adjust the stack pointer upon return

Example

A NetWare call to a driver's initialization routine would have the following syntax:

```
long   DriverInitialize  (
                long  ModuleHandle ,
                long  ScreenHandle ,
                byte *CommandLine ,
                long  reserved0 ,
                long  reserved1 ,
                long  LoadableModuleFileHandle ,
                long (*ReadRoutine) ( ) ,
                long *CustomDataOffset ,
                long  CustomDataSize  ) ;
```

Some NetWare support routines require use of parameters passed on the stack when *DriverInitialize* is called. These values should be saved by the initialization procedure for later reference. For example, the driver's initialization routine must call the NetWare support routine *ParseDriverParameters*, which appears to the driver as shown below. To pass the command line pointer and screen handle, the driver's initialization routine must retrieve them from the stack.

```
long  ParseDriverParameters  (
            struct IOConfigurationStructure  *DriverConfiguration ,
            long  reserved0 ,
            struct AdapterOptionStructure *AdapterOptions ,
            long  reserved1 ,
            long  reserved2 ,
            long  NeedsBitMap ,
            byte *CommandLine ,
            long  ScreenHandle  ) ;
```

Given the syntax shown above, a typical initialization routine could call *ParseDriverParameters* as shown below.

```
DriverInitialize   proc

   CPush
   mov    ebp, esp
   pushfd
   cli
     •
     •
     •
   push   [ebp + Parm1]                              ;ScreenHandle
   push   [ebp + Parm2]                              ;CommandLine
   push   NeedsIOPort0Bit OR NeedsInterrupt0Bit      ;Parse for port and int
   push   0                                          ;reserved
   push   0                                          ;reserved
   push   OFFSET AdapterOptions                      ;Adapter Options Structure
   push   0                                          ;reserved
   push   OFFSET DriverConfiguration                 ;IOConfiguration

   call   ParseDriverParameters

   add    esp, 8 * 4
```

Note that the parameter values passed to *ParseDriverParameters* are pushed in reverse order. After the initialization routine calls *ParseDriverParameters*, it adjusts the stack pointer by 8 * 4.
Eight *push* instructions times four bytes (each *push* is one *dword*).

In the example, Parm0, Parm1, ...etc. are defined as follows:

```
ParmOffset      equ  20
Parm0           equ  ParmOffset + 0
Parm1           equ  ParmOffset + 4
Parm2           equ  ParmOffset + 8
Parm3           equ  ParmOffset + 12
Parm4           equ  ParmOffset + 16
Parm5           equ  ParmOffset + 20
Parm6           equ  ParmOffset + 24
Parm7           equ  ParmOffset + 28
Parm8           equ  ParmOffset + 32
```

ParmOffset represents the 20 bytes normally pushed on the stack when a C-style call is made (4 by the call instruction, and 16 when saving EBX, EBP, ESI, and EDI). Defining the stack offsets this way is one method that can simplify the retrieval of parameters off the stack. However, the driver can use any method preferred.