3

Data Structures, Tables, and Variables

Data Structures and Tables

MSL drivers must create and/or maintain several data structures and tables in order to interface with the NetWare SFT III operating system. Additional structures are optional and can be defined and used by the driver as needed.

Most MSL drivers will require the following tables and structures:

- *DriverStatistics Table*
- *IOConfigurationStructure*
- *AdapterOptionStructure*
- *AESEventStructure*
- *TimerDataStructure*

A brief description of these tables and structures is provided below. The following pages then provide detailed information.

The *DriverStatistics* table contains various diagnostic counters that the driver must maintain. Each physical adapter will have a corresponding statistics table.

The *IOConfigurationStructure* contains information about the adapter's hardware configuration. The structure is required when the driver calls various operating system support routines.

The *AdapterOptionStructure* allows the driver to provide lists of valid configuration options available for the adapter's hardware. These lists are used by the *ParseDriverParameters* support routine to prompt for and validate configuration information entered from the load command line or interactively from the operator console. The *ParseDriverParameters* routine uses the information to fill out the driver's *IOConfigurationStructure*.

The *AESEventStructure* and *TimerDataStructure* are used to schedule *callback* events to a specified driver routine after a designated interval. For example, a driver routine could be scheduled for callback in order to monitor for and recover from timeout conditions or to perform retry operations at a later time. The *AESEventStructure* is used to schedule callbacks to driver routines that must run at *process time*. The *TimerDataStructure* is used to schedule *interrupt time* callbacks.

DriverStatistics Table

The *DriverStatistics* table is a structure containing various diagnostic counters used to monitor operations related to the MSL driver or adapter. The format of the statistics table is strictly defined and is illustrated below. A description of each field follows the sample.

The table is divided into two general sections: the generic (or standard) MSL statistics required by NetWare, and the custom statistics defined by the driver. Your driver must maintain *all counters* in the table.

The driver must provide external processes (upper layer applications) access to the statistics table information through the *DriverControl* procedure, *GetMSLStatistics*. When called, this procedure creates a copy of the statistics table in a buffer provided by the caller. Refer to Chapter 4, "MSL Driver Procedures," for more information on this routine.

```
*   DriverStatistics            db   0 dup (?)

    StatisticsMajorVersion      db   01
    StatisticsMinorVersion      db   00
    NumGenericCounters          dw   (GenericEnd - GenericBegin) / 4
    NotSupportedMask            dd   00000000000000000011111111111111b
 *  GenericBegin                db   0 dup (?)
       TransmitPacketCount      dd   0
       ReceivePacketCount       dd   0
       TransmitBurstPacketCount dd   0
       MSLRejectPacketCount     dd   0
       TransmitMsgCount         dd   0
       ReceiveAckCount          dd   0
       ReceiveMsgCount          dd   0
       TransmitAckCount         dd   0
       TransmitHoldCount        dd   0
       ReceiveHoldCount         dd   0
       OSHoldMsgCount           dd   0
       OSCallBackCount          dd   0
       OSRejectMsgCount         dd   0
       ServerCommErrorCount     dd   0
       ReceiveErrorCount        dd   0
       TransmitErrorCount       dd   0
       ReceiveEmergencyCount    dd   0
       RetryTxCount             dd   0
 *  GenericEnd                  db   0 dup (?)

    NumCustomCounters           dw   (CustomEnd-CustomBegin)/4
 *  CustomBegin                 db   0 dup (?)
       CustomCounter1           dd   0
         .
         .
         .
       CustomCounterN           dd   0
 *  CustomEnd                   db   0 dup (?)
 *  CustomStrings               db   0 dup (?)



 *  These fields are 0 bytes in length and function as labels.
    (This syntax is a feature of the Phar Lap assembler.)
```

Statistics Table Field Descriptions

Offset	Name	Bytes	Description
00h	StatisticsMajorVersion	1	This field contains the major version number of the statistics table. The version number is controlled by Novell and is currently v1.00; therefore, 1 is the major version number.
01h	StatisticsMinorVersion	1	This field contains the minor version number of the statistics table. This version number is controlled by Novell and is currently v1.00; therefore, 00 is the minor version number.
02h	NumGenericCounters	2	This field contains the number of generic counters present in the statistics table (but not necessarily supported or used). Currently, this field should be set to 18 (decimal).
04h	NotSupportedMask	4	This field contains a bit mask indicating which *generic* counters of the statistics table are implemented. The 18 most significant bits correspond to each of the generic counters (with the most-significant bit matching *TransmitPacketCount*). If the bit is 0, the counter is supported; if the bit is 1, the counter is not supported. The remaining 14 bits should be padded with ones.
08h	TransmitPacketCount	4	Total number of *message packets* successfully transmitted by this MSL adapter.
0Ch	ReceivePacketCount	4	Total number of *message packets* successfully received by this MSL adapter.
10h	TransmitBurstPacketCount	4	Total number of *burst message packets* transmitted by the MSL adapter. Burst packets are constructed by the *DriverBuildSend* routine after *DriverISR* receives a message acknowledgement. Typically these packets contain multiple messages but might contain only 1 message. (see Chapter 4)
14h	MSLRejectPacketCount	4	Number of packets MSL rejects.
18h	TransmitMsgCount	4	Total number of messages transmitted.
1Ch	ReceiveAckCount	4	Total number acknowledgments received. (should correspond to *TransmitMsgCount*)
20h	ReceiveMsgCount	4	Total number of messages received.
24h	TransmitAckCount	4	Total number of acknowledgments transmitted. (should correspond to *ReceiveMsgCount*)
28h	TransmitHoldCount	4	Total number of hold notifications transmitted.
2Ch	ReceiveHoldCount	4	Total number of hold notifications received.
30h	OSHoldMsgCount	4	Total number of messages that the OS held off before accepting. (This is the number of times *ReceiveServerCommPointer* returns a status code of 2 or 3)

Statistics Table Field Descriptions
-(continued)-

34h	OSCallBackCount	4	Total number of times the OS has requested to be called back after the driver copied the message to system memory. (This is the number of times *ReceiveServerCommPointer* returns a status code of 1)
38h	OSRejectMsgCount	4	Total number of messages rejected (ignored) by the OS. (This is the number of times *Receive-ServerCommPointer* returns a status code of 4)
3Ch	ServerCommErrorCount	4	Total number of times the driver called the OS procedure *ServerCommDriverError*.
40h	ReceiveErrorCount	4	Total number of receive errors.
44h	TransmitErrorCount	4	Total number of transmit errors.
48h	ReceiveEmergencyCount	4	Total number of emergency notification packets received from the other server.
4Ch	RetryTxCount	4	Number of transmit retries.
50h	NumCustomCounters	2	This field contains the number of custom counters defined by the driver.
52h . . .	CustomCounter1 . . .	4 each	These fields contain custom counters that can be defined for the specific needs of the MSL driver or adapter design. Each custom counter must have a corresponding string in the custom strings area (defined below).

??h	CustomStrings	?	The CustomStrings area provides diagnostic strings that correspond to the custom counters. The first word of the CustomStrings area contains the size of the area in bytes. Each string must be null terminated. The table of strings is terminated with two nulls.
			
			`CustomStrings db 0 dup (?)`
			`CustomStringsSize dw EndStrings-CustomStrings`
			`db 'Custom String 1', 0`
			`db 'Custom String 2', 0`
			`db 'Custom String 3', 0`
			`.`
			`.`
			`.`
			`db 'Custom String N', 0`
			`db 0, 0`
			`EndStrings db 0 dup (?)`

IOConfigurationStructure

The *IOConfigurationStructure* defined by NetWare contains fields describing information about the adapter's hardware configuration including I/O ports, memory decode addresses, interrupts, and DMA channels. The configuration structure is shown on the following page. A description of each field follows the example.

The MSL driver uses the structure primarily during initialization to reserve file server hardware resources.

The following OS support routines require the structure:

- *ParseDriverParameters*
- *RegisterHardwareOptions*
- *DeRegisterHardwareOptions*
- *RegisterServerCommDriver*

The driver calls *ParseDriverParameters* to fill in the fields of the structure using information entered from the load command line and/or interactively from the operator console. All fields of the structure must be zeroed prior to calling *ParseDriverParameters*, unless noted otherwise. (Chapter 5 describes the *ParseDriverParameters* support routine in detail.)

Once the configuration table is filled in, the driver calls *Register-HardwareOptions* to reserve the configuration options with the OS. The driver *must not modify any field* in the configuration structure after calling *RegisterHardwareOptions*.

The driver must provide external processes (upper layer applications) access to the configuration table information through the *DriverControl* procedure, *GetMSLConfiguration* (see Chapter 4). When called, this procedure creates a copy of the configuration table in a buffer provided by the caller.

```
IOConfigurationStructure    struc

     CLink                  dd     ?
     CFlags                 dw     ?
 *   CSlot                  dw     ?

 *   CIOPort0               dw     ?
 *   CIOLength0             dw     ?
 *   CIOPort1               dw     ?
 *   CIOLength1             dw     ?

 *   CMemoryDecode0         dd     ?
 *   CMemoryLength0         dw     ?
 *   CMemoryDecode1         dd     ?
 *   CMemoryLength1         dw     ?

 *   CInterrupt0            db     ?
 *   CInterrupt1            db     ?

 *   CDMAUsage0             db     ?
 *   CDMAUsage1             db     ?

     CIOResourceTag         dd     ?

     CConfiguration         dd     ?
     CCommandString         dd     ?
     CLogicalName           db     18 dup (?)
     CIOReserved            db     16 dup (?)

IOConfigurationStructure    ends



DriverConfiguration         IOConfigurationStructure
                                  <0,0,0,
                                   0,0,0,0,
                                   0,0,0,0,
                                   0FFh,0FFh,
                                   0FFh,0FFh,
                                   0,0,0,0,0>
```

* These values are configurable from the command line and/or interactively from the server console at load time.

Configuration Field Descriptions

Offset	Name	Bytes	Description
00h	CLink	4	Reserved by NetWare. Drivers should not change this field.
04h	CFlags (sharing flags)	2	Setting these bits indicates that the MSL adapter can share I/O ports, memory ranges, interrupts, and/or DMA channels. (Note: Set bit 0 to 0.) This field must be initialized *before* calling the *ParseDriverParameters* routine. Set this field to zero unless the hardware options are shared on all the adapters using this driver. For shared I/O device addresses, interrupts, etc., place the following value in this field: IODetached 01h IOSharablePort0Bit 02h IOSharablePort1Bit 04h IOSharableMem0Bit 08h IOSharableMem1Bit 10h IOSharableInt0Bit 20h IOSharableInt1Bit 40h IOSharableDMA0Bit 80h IOSharableDMA1Bit 100h
06h	CSlot	2	If an MSL adapter is running in an MCA or EISA machine, this field holds the slot number where the adapter is installed. (zero if not used)
08h	CIOPort0	2	This field contains the primary base I/O port for the MSL adapter. (zero if not used)
0Ah	CIOLength0	2	This field contains the number of I/O ports starting at *CIOPort0*. (zero if not used)
0Ch	CIOPort1	2	This field contains the secondary base I/O port for the MSL adapter. (zero if not used)
0Eh	CIOLength1	2	This field contains the number of I/O ports starting at *CIOPort1*. (zero if not used)
10h	CMemoryDecode0	4	This field contains the *absolute* address of primary shared memory used by the MSL adapter. (zero if not used)
14h	CMemoryLength0	2	This field contains the amount of memory (in paragraphs) that the MSL adapter uses, starting at *CMemoryDecode0*. (zero if not used)
16h	CMemoryDecode1	4	This field contains the *absolute* address of secondary shared memory used by the adapter. (zero if not used)

Configuration Field Descriptions

-(continued)-

1Ah	CMemoryLength1	2	This field contains the amount of memory (in paragraphs) that the MSL adapter uses, starting at *CMemoryDecode1*. (zero if not used)
1Ch	CInterrupt0	1	This field contains the primary interrupt number used by the adapter. (FFh if not used)
1Dh	CInterrupt1	1	This field contains the secondary interrupt number used by the adapter. (FFh if not used)
1Eh	CDMAUsage0	1	This field contains the primary DMA channel used by the MSL adapter. (FFh if not used)
1Fh	CDMAUsage1	1	This field contains the secondary DMA channel used by the MSL adapter. (FFh if not used)
20h	CIOResourceTag	4	This field contains the resource tag with a *IORegistrationSignature* acquired by the driver during initialization. (see *AllocateResourceTag*)
24h	CConfiguration	4	Reserved. MSLs should not use this field.
28h	CCommandString	4	If the driver needs to append something to the command line or replace the default command line in the AUTOEXEC.NCF file for the I/O Engine (IOAUTO.NCF), this field will contain a long pointer to the new or additional command line string. If the driver does not use this option, the field must be cleared. If this field is used, then bits 9 and 10 of *CFlags* need to be set appropriately.
2Ch	CLogicalName	18	MSLs should not use this field. This field contains the logical name of the MSL driver, if it is given one at load time. For example: `load <driver> name="________"`
3Eh	CIOReserved	16	Reserved for the MSL's use.

AdapterOptionStructure

The *AdapterOptionStructure* is defined in the MSL.INC file and is shown below. In order to use the NetWare support routine *ParseDriver-Parameters* to parse the load command line, an MSL driver must maintain a single instance (or more if more than one adapter type is supported by the same driver) of this structure.

The *AdapterOptionStructure* serves as a template defining the available choices for various adapter configuration options. *ParseDriver-Parameters* uses this template to parse the load command line, query the operator for any required options not found on the command line, validate the selected values, and fill in the associated fields in the *IOConfigurationStructure*.

```
AdapterOptionStructure  struc
   IOSlot               dd     ?
   IOPort0              dd     ?
   IOLength0            dd     ?
   IOPort1              dd     ?
   IOLength1            dd     ?
   MemoryDecode0        dd     ?
   MemoryLength0        dd     ?
   MemoryDecode1        dd     ?
   MemoryLength1        dd     ?
   Interrupt0           dd     ?
   Interrupt1           dd     ?
   DMA0                 dd     ?
   DMA1                 dd     ?
AdapterOptionStructure  ends
```

Each field in the structure is a pointer to a length-preceded list of allowable options for that field. The first option in the list is used as the default value. Each list assumes the following form:

```
List    dd   n          ;number of entries
        dd   entry1      ;first (default) value
        dd   entry2
        .
        .
        dd   entryn      ;last available value
```

If entries are not used in the *AdapterOptionStructure*, the pointer to the associated list should be set to zero. The fields are explained in detail under the *IOConfigurationStructure* description in the previous section.

Refer to the *ParseDriverParameters* description in Chapter 5, "NetWare SFT III Support Routines," for additional information on the use of this structure.

AESEventStructure

The *AESEventStructure* is required to schedule *process level callbacks* to a specified driver routine after a designated interval. For example, a driver routine could be scheduled for callback in order to monitor for and recover from transmit timeout conditions, or to perform retry operations at a later time.

The OS support routines, *ScheduleSleepAESProcessEvent* and *ScheduleNoSleepAESProcessEvent* described in Chapter 5, are used to schedule the callback events for either *blocking* or *non-blocking* process level driver routines.

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

Field Name	Description
AESLink	Used internally by the NetWare OS; do *not* modify this field.
AESWakeUpDelayAmount	The amount of time in system clock ticks (1 tick ≈ 1/18 second) before the callback procedure is invoked. Generally, this interval should be small enough to provide reasonable recovery time, but not so small as to affect overall server performance.
AESWakeUpTime	Used internally by the NetWare OS; do *not* modify this field.
AESProcessToCall	A pointer to the routine that will be called once for each *ScheduleAESProcessEvent* call.
AESRTag	Resource tag with an *AESProcessSignature* obtained by the MSL driver during initialization. This allows the OS to track the AES resource.
AESOldLink	Maintained for backward compatibility.
MessageTimeOutTime (optional)	Set this field to the value of *ServerCommACKTimeOut* when beginning a message timeout sequence. This value is the maximum time (in ticks) you should wait for the other server's acknowledgment before calling *ServerCommDriverError*.
AdapterTimeOutTime (optional)	This field is used for adapters that support the transmit complete feature. When the driver initiates a transmission, it should set this value to the maximum time (in ticks) to wait for that transmission to complete. This can be used to detect a "dead" adapter.

TimerDataStructure

The *TimerDataStructure* is required to schedule *interrupt level callbacks* to a specified driver routine after a designated interval. The OS support routine used to schedule the callback is *ScheduleInterrupt-TimeCallBack*. This routine adds an event to the list of events that will be called by the timer tick interrupt handler.

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

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

Field Name	Description
TCallBackProcedure	Pointer to the procedure to be called by the timer interrupt handler. When the procedure is called, interrupts are disabled.
TCallBackEBXParameter	The value EBX should contain when the call back procedure is invoked.
TCallBackWaitTime	The amount of time (in ticks) before the callback procedure is invoked.
TResourceTag	Resource tag with a *TimerSignature* acquired by the driver for interrupt time callbacks. (see *AllocateResourceTag*)

Note: The four fields described above are not changed by the operating system. If the driver reschedules another callback, it does not need to reinitialize these fields.

Global Data Variables

This section describes the global data variables that have special meaning for MSL drivers. All variables described in this section are external to the driver. The driver will either need to initialize, maintain, or access these variables during the course of its operation.

MaximumCommDriverDataLength: dword

The driver must set this variable during initialization so that the operating system knows the maximum message size (not including the *message header*) that the driver can transmit.

PacketSizeNowAvailable: dword

The OS maintains this variable to let the driver know the size of the next message (not including the *message header*) that the OS has queued for transmission. This value may be positive, zero, or negative. A negative value indicates that the OS has no messages queued for transmission. A value of zero indicates a *message header only* with no *message data*.

PacketSizeDriverCanNowHandle: dword

The driver maintains this variable to let the OS know whether it can call the *DriverSend* or *DriverBuildSend* routines to send the next message. This value may be positive, zero, or negative. A negative value indicates the driver can send no more messages (typically until an acknowledgement is received from the other server for a previous message). A value of zero indicates the driver can send a *message header only* with no *message data*.

There are several cases when the OS does not check this variable before calling the driver to send a message:

Case 1. When the MSL driver initially registers with the mirrored server interface by calling *RegisterServerCommDriver*, the OS assumes that the driver is ready to send a message.

Case 2. When the OS sends an initial "I'm alive" message and receives an acknowledgement, it will send the next message in the initial link protocol between the two servers without checking this variable. The OS assumes that if the MSL driver has received an acknowledgement from the initial message, it should be ready to transmit another message.

Case 3. The OS sends an initial "I'm alive" message and times out waiting for the acknowledgement. The OS then switches into a "listening" state. Upon reception of an "I'm alive" message, the OS will send a reply message without checking this variable. The OS assumes that if the first message has timed out, the MSL should be ready to transmit another message.

ServerCommACKTimeOut: dword

The OS sets this variable to the maximum time in ticks the driver should wait for a message acknowledgement. The MSL driver uses this value in conjunction with the *DriverTimeout* routine to determine if the mirrored server-to-server link is still active.

When the MSL driver sends a message, it initializes a counter to the value of *ServerCommACKTimeOut*. The *DriverTimeout* routine (which is called back at 1 tick intervals) will decrement the counter. If the value becomes zero before the acknowledgement is received, the timeout routine should notify the OS that the link between the mirrored servers is no longer valid by calling the *ServerCommDriverError* routine.

The OS may dynamically change the *ServerCommACKTimeOut* value; therefore, each time a new message timeout is started, the counter should be reinitialized to the current *ServerCommACKTimeOut* value.

Indirect OS Calls

The following global variables are defined and maintained by the NetWare SFT III operating system. These variables contain pointers to specific OS procedures that the MSL driver must access. This section provides a brief description of these indirect calls. Each procedure is then described in detail in Chapter 5.

GetNextPacketPointer: dword

This variable contains a pointer to the current OS procedure used to get the next message (or group of messages) that the OS has queued for transmission. The MSL driver must call this procedure after it receives an acknowledgement. This procedure initiates a possible multi-message packet building sequence.

ReceiveServerCommPointer: dword

This variable contains a pointer to the current OS procedure used to notify the OS when a message is received from the other server. This

procedure then returns a completion code indicating to the driver what action to take for the message.

SendServerCommCompletedPointer: dword

This variable contains a pointer to the current OS procedure used to notify the OS when any message acknowledgements are received from the other server.