Novell®

Everything's Connected™

Novell Labs

ATM ODI HSM Specification

Version 1.0

ATM ODI HSM Specification
Documentation Part Number: 107-000083-001
June 2, 1997

disclaimer

trademarks

Novell, Inc.
122 East 1700 South
Provo, UT 84606

Document Release Date: December 23, 1996
Documentation Part Number: 107-000083-001

Novell®
Everything's Connected™

Contents

1 Overview

2 ATM TSM Components

3 ATM HSM Interface

4 ATM HSM Definitions

5 Virtual Connection Establishment Flow Diagrams

6 Include Files

A Glossary

B References

Index

Novell®
Everything's Connected™

About This Guide

The page numbering format in this document is:

<CHAPTER NUMBER> <HYPHEN> <PAGE NUMBER>

An example and explanation follows:

1-1

2-23

4-45

and so on.

Page numbers are sequential from the beginning of the book and do not restart at the beginning of each new chapter. This means that the first page of chapter 2 may be on page 23, the first page of chapter 4 may be on page 45.

This numbering format tells you immediately what chapter and page number you are currently looking at.

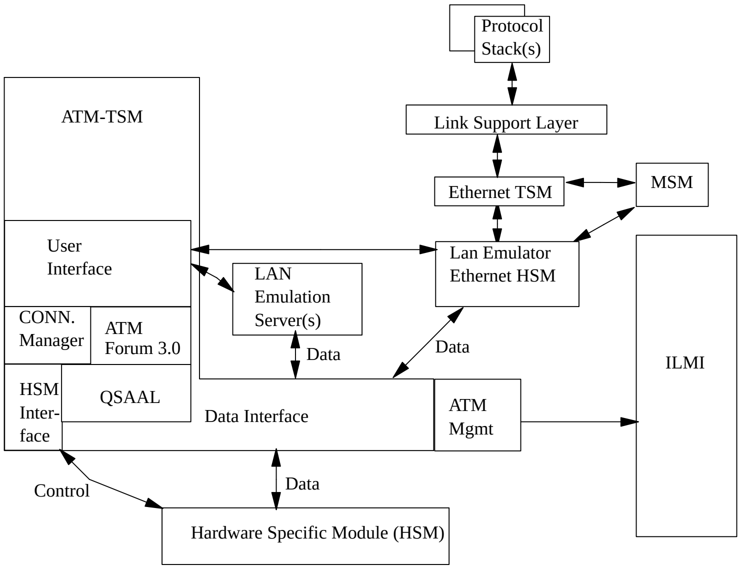
Novell®
Everything's Connected™

chapter 1 Overview

Introduction

This document describes the operation and Application Programming Interfaces (APIs) of the Asynchronous Transfer Mode-Hardware Specific Module (ATM-HSM) portion of the Novell Open Data-Link Interface (ODI) for ATM. The ATMTSM provides two groups of APIs, one is used by ATM Hardware Specific Modules (HSMs), and one is used by User Applications that require direct access to ATM services. The HSM API sits at the bottom of the ATMTSM, while the User Application API sits at the top. An overview diagram is shown below.

Figure 1-1
ATM Environment Overview



What is ATM

ATM is the base technology for the next generation of global communications, spanning diverse applications and providing a variety of interface speeds and distances. Originally conceived as the switching technology for Broadband ISDN, ATM now has taken on a wide use for immediate deployment in local and wide area networks of all types.

The importance of ATM goes beyond pure technology. It is one of the few networks that can provide real-time and quality of service guarantees required for the new multimedia data applications. There is consensus across diverse industry participants of voice, video, and data communications that ATM is the enabling technology that brings us into the information age.

ATM is not a shared access technology so it is different than conventional LAN topologies such as Ethernet, Token-Ring and FDDI. Shared access technologies consist of a common wire used by all attached nodes to communicate. Adding extra nodes reduces the bandwidth available to the existing nodes. This sets an upper limit to the number of nodes that can be attached to a shared access network like Ethernet. ATM does not suffer from this limitation so the number of nodes supported can be much larger.

Cell Switching Systems

The backbones of an ATM network are high-speed cell switching systems. These systems, called switches, are interconnected by physical transmission links. End stations are directly connected to a switch. End stations create connections between end stations much like X.25. The end stations send data onto the network as fixed length cells. Each cell is 53 bytes long. Larger frames (up to 64 Kilobytes) are sliced into cells. The cell containing the end of the frame can be padded. The sending end stations are responsible for proper pacing of cells. The pacing is based on a contract that can differ from connection to connection, the direction of the connection, and the Quality of Service (QoS) required. Voice and video can use Constant or Variable Bit Rates, while data is more likely to use Available or Unspecified Bit Rates. All information, whether it is voice, video or data uses the same cell structure and size. Transport protocols must be used to guarantee delivery.

The path for a given connection is determined during connection establishment. This means that per-cell routing is not required. End stations can create point-to-point or point-to-multipoint connections. For point-to-multipoint connections, a switch replicates the data and delivers a copy to each end station, or leaf on the multipoint connection.

ATM and NetWare

Unlike conventional LANs, ATM does not necessarily use MAC and LLC headers. Thus it cannot rely upon the Link Support Layer (LSL) software to do the per packet demultiplexing.

ATM is connection oriented so other entities are required to establish the connections. This ATM HSM specification document does not provide the needed information explaining User Applications used to create Switched Virtual Connections (SVCs) or Permanent Virtual Connections (PVCs). The scope of this document is limited to developing the ATM driver for NetWare. References to the User Interface are provided only for informational purposes.

ATM and HSM

ATMTSM (Topology Specific Module) is the (non-ISO) layer residing between the ATM HSMs and the ATM User Applications. The ATMTSM separates the connection management functions from the hardware specific tasks. This separation means that an HSM does not need to know whether VCs (Virtual Connections) are permanent or switched, nor does it need to know whether they are incoming or outgoing.

The ATMTSM enables and disables the VCs individually because the hardware or firmware is usually responsible for maintaining the VC contracts, such as the rate of transmission. Conversely, the User Applications do not need to know any of the hardware-related issues. When establishing a VC, all the User Application needs to know about the hardware is its interface identifier. The ATMTSM registers an adapter board with the LSL for each interface that the HSM registers with the ATMTSM. Thus, the interface identifier is an adapter board number.

A priority is assigned to each VC. The HSM should be able to make assignments for receive resources based on the VC priority. When the CPU is temporarily busy processing previously received frames, the HSM can advance higher priority frames ahead of lower ones. If the CPU gets behind to the point where the HSM has a high volume of received frames in queues, it should discard low priority VC frames, in favor of the high priority VCs. There are eight priority levels and the higher priorities are typically used for real-time data.

To optimize performance, the HSM is VC and ECB (NetWare buffer) aware, for both the transmit and receive interface. Both the HSM and the ATMTSM maintain their own Adapter Data Space (ADS). During the registration process, the ADS addresses are exchanged. The ATMTSM will pass the HSM's ADS address when it passes control for interrupts and/or polls. Procedures to lock/unlock interrupts and/or multiple processors are provided by the ATMTSM. The HSM will have to determine when locks are needed and call the proper lock/unlock procedures.

Novell®
Everything's Connected™

chapter 2 ATM TSM Components

Introduction

The ATMTSM contains the following functional components:

- Q.2931 Signaling
- Q.SAAL1
- Connection Manager
- User Interface
- HSM Interface
- Data Interface
- ATM Management
- Data Monitor/Tracing

Q.2931 Signaling

Q.2931 is an ITU-defined call establishment and release protocol. The ATMTSM contains the signaling logic as defined by the ATM Forum's User-Network Interface (UNI 3.0) implementation agreement. Signaling is the protocol used to establish connections with other end stations across one or more ATM switches.

Q.SAAL1

Q.SAAL1 is a transport-like point-to-point link protocol. The Q.2931 signaling protocol elements rely on links that guaranty delivery between the end stations and switches, and between the switches themselves. A link is established between the end station and the switch, as well as between each pair of switches. These links use a well-known PVC Identifier (PVC 5). The Q.SAAL1 protocol runs across each individual link.

Connection Manager

The connection manager is the core functionality of the ATMTSM. The connection manager is responsible for managing the establishment and release of the virtual connection between the User Application interface and the HSM. It is implemented as a finite state machine, that receives control indicators from the User Interface, the HSM Interface, and the Q.2931.

Connection establishment between the ATMTSM and HSM is always done with two-way handshaking. Connection release, on the other hand, is a unilateral command that can originate from the User Application, the ATMTSM or the HSM. After the first handshake, a connection can always be released, even before it is completely established and enabled.

User Interface

Note The User Application Interface is documented here for informational purposes only. Driver developers do not currently have access to the User Interface hooks.

After the User Application registers with the ATMTSM for an individual interface (board), the User Interface allows for the setup of Permanent Virtual Connections (PVCs) and incoming or outgoing Switched Virtual Connections (SVCs). Leafs can be added to outgoing SVCs. During connection establishment, the User Application provides a User VC handle and separate Data and Control entry points. The ATMTSM returns an ATMTSM VC handle to the User. Once the VC has been established successfully, the User and ATMTSM will exchange data, qualified only by the handles.

Data passed to the User is accompanied by the User VC handle, and data to the ATMTSM is accompanied by the ATMTSM VC handle. The User determines the value of the User VC handle, probably the address of a User Connection Control Block. The ATMTSM determines the value of the ATMTSM VC handle. Additional interfaces are available for the User Application to manage the ATM addresses.

HSM Interface

The HSM Interface allows one or more HSMs to register and deregister one or more ATM boards. The ATMTSM allows the HSM to register its hardware separately and more than once.

This enables an HSM to partially register its hardware, learn about the rest of the hardware, deregister the partial hardware, and re-register all the required hardware it needs. After successful registration, the ATMTSM must issue a Start Interface command. After that command, the ATMTSM requests the HSM to enable or disable certain VCs. The ATMTSM either knows the VCI for the PVC, or learns the VCI for the SVC through the Q.2931 signaling protocol. During VC enable, the ATMTSM exchanges handles with the HSM (there is an HSM VC handle as well as an ATMTSM VC handle). These handles are passed between the ATMTSM and HSM. The HSM determines the value of the HSM VC handle, generally the address of a VC oriented Control Block. The ATMTSM determines the value of the ATMTSM VC handle.

Data Interface

The data interface is responsible for the data exchange between the HSM and the User Application once the connection manager has established the VC. Because the HSM passes the HSM VC handle with incoming data to the ATMTSM, the ATMTSM can, within a few cycles, switch ownership of the receive ECB, locate the User VC handle, maintain statistics, and pass the data along to the User. Outgoing data follows an even faster route through the ATMTSM, because ECB ownership is not changed.

All the data interfaces between the User Application and the ATMTSM are done with indirect calls. Indirect calls are also used for data interface between the HSM and ATMTSM . This allows for remote monitoring and possibly filtering applications to preview the incoming and outgoing data on a per VC basis, without negatively affecting the performance for the normal data path.

ATM Management and Address Registration

ATM Management supports the ATM Forum UNI 3.0 ILMI. This component’s main responsibility is to manage the registration of the ATM addresses. On a private switch, an ATM address is composed of a Network part and an End Station Identifier (ESI). The switch defines and manages the Network part of the address, while the end-station manages the ESIs. The User Interface has APIs for the ESI management.

Novell®
Everything's Connected™

chapter 3

ATM HSM Interface

Introduction

This section describes the issues that the developer of an ATM driver (HSM) needs to consider. The HSM is mostly concerned with the hardware and firmware issues of the ATM interface board. The ATMTSM performs many of the common interface functions between NetWare and the HSM. The ATM link speed can be very high, anywhere from 25 Megabits to 625+ Megabits, so it must be assumed that the majority of the boards demultiplex VCs with the hardware and firmware. Therefore, the ATMTSM/HSM interface is “VC aware”.

The NetWare ODI Developer’s Guide for NetWare Server Driver Hardware Specific Modules (part number 107-000007-001) provides information about how to build an HSM. When referencing that specification, bear in mind that the ATMTSM does not utilize the Adapter Data Space defined for NetWare LAN drivers in the Media Support Module (MSM).

The HSM can use certain CMSM functions. These functions must not require the ODI LAN adapter or Frame Data Space.

The few MSM functions that the ATM HSM needs are currently provided by the ATMTSM. The ATMTSM and HSM do not share a common Adapter Data Space, they each have their own. When registering the HSM with the ATMTSM, the ATMTSM acquires a board number. This board number is associated with an MLID configuration table. The configuration table is not made available to the HSM.

All interfaces defined by the ATMTSM for the HSM are ANSI-C Language based. Refer to the appropriate reference manuals for the target platform. For compiling ATM drivers on NetWare 4.11, refer to the WATCOM C Compiler manuals.

Note The term “HSM” is synonymous with driver.

This section deals with the following issues:

- Building the driver
- Adding the Specification Version String
- Parsing the LOAD driver command line
- Optionally loading driver firmware
- Registration
- Deregistration
- HSM Control interfaces (bidirectional)
- Get/Return buffers (ECBs)
- Data interfaces
- Interrupts
- Polling / Periodic Calls
- Statistics
- Additional interfaces

Building the driver

Just as with regular ODI drivers, the ATM HSM driver name must have the suffix “LAN”. The NetWare ODI Server Driver Specification provides information about how to build a NetWare ODI driver.

Specification Version String

The driver must also have an embedded specification version string to identify the version of the ATM HSM specification used. The current version is 1.0 for the ATM driver specification. The 1.00 in the specification string below signifies version 1.0.

The following string must be added exactly as shown to the ATM HSM where the global variable declarations are made:

```
MEON_STRING HSMSPEC[]=“HSM_ATMSPEC_VERSION: 1.0”;
```

Note that only one space is required between the colon and the one.

MEON_STRING is defined in ODI.H.

Parsing the LOAD driver command line

Often, a driver requires operator-configurable parameters. Examples of configurable parameters are the port number, shared memory address and size, interrupt number, DMA channel, slot number, and so on. The driver is responsible for parsing the command line and passing the various configurable parameters to ATMTSM during registration. The driver may opt to use CMSM ParseDriverParameters or PARSER.NLM. The PARSER.H include file describes how the table driver interfaces with PARSER.NLM.

Optionally loading driver firmware

A procedural interface (*atmtsm_get_firmware*) is provided that allows the driver to acquire the memory address and the length of its firmware. The HSM is responsible for actually loading the firmware onto the board.

Registration

After the driver has been successfully loaded, the HSM driver needs to register with the ATMTSM. The atmhsmif.h include file contains all the definitions the driver needs to perform this registration. This file defines a structure called ATMTSM_REGISTER_HSM_CONTROL.

The HSM must fill out all the fields in this structure and call *atmtsm_register_hsm*, passing the address of the structure. If the registration is successful, the ATMTSM returns a zero value. If the registration fails, the ATMTSM returns a non-zero value, and logs an appropriate error message on the console. Registration must be done during driver loading and initialization.

During registration, the ATMTSM will return the board number as well as the ATMTSM Adapter Data Space (ADS) handle. The HSM must also provide the ATMTSM with either the size of the HSM Adapter Data Space or its handle. If the HSM passes a non-zero ADS size, the ATMTSM will allocate the HSM ADS, and return the HSM ADS handle.

After the HSM has successfully registered, it must register its hardware with the ATMTSM. If the HSM needs some of the hardware to find out about the rest of the hardware, it may register/deregister the hardware multiple times.

After successful registration of the HSM and hardware, the driver must pass a START INTERFACE control command to the ATMTSM. The ATMTSM does not attempt to verify any contract, nor enable any VCs until after this command is received. The driver can use the time between the registration and START INTERFACE to do such initialization as loading firmware, setting up interrupts, preparing receive and transmit queues, and so on.

Note The driver must register each ATM interface individually.

Fields passed between HSM and ATMTSM during registration

The following table shows the various fields that are passed between the HSM and the ATMTSM during registration. The table is only used by the ATMTSM during the registration process, thus, the HSM might build this table on the stack:

version

The HSM must insert the ATMTSM_HSM_VERSION constant. This constant is also defined in the atmhsmif.h include file. The version allows the ATMTSM to verify that the HSM is built with the same version. When the HSM is built with an older version, the ATMTSM might be able to support that older driver, depending on whether the changes warrant backward compatibility. This is determined on a case-by-case basis.

hsm_nlm_handle_ptr

This field contains the stack-address of the NetWare NLM handle that is passed to the driver when its initial “start” entry point is being called by NetWare (immediately after the driver has been loaded). The ATMTSM uses this handle to allocate various resources for the HSM. This includes reading the firmware, when defined.

This field is a pointer to the first argument passed to the initial function. For example:

```
void hsm_init( void *handle)

{

...

reg.hsm_nlm_handle_ptr=&handle;

...}
```

hsm_control_proc

This field defines the entry point that the ATMTSM calls whenever it has Control Information for the HSM. The type and format of this information is defined later in this section (see ATMTSM to HSM Control Commands). The entry point must be of C language type ATMTSM_HSM_CONTROL_PROC.

hsm_send_proc

This field defines the entry point that the ATMTSM calls whenever it has a data frame for the HSM to send. The entry point must be of C language type ATMTSM_HSM_SEND_PROC.

card_name

This field is a pointer to a string that contains the (long) descriptive name of the card. The name can contain blanks and must be terminated by a zero. The ATMTSM will copy this name to a local structure. The maximum length is MAX_CARD_NAME.

driver_name

This field is a pointer to a string that contains the (short) name of the driver. The name must be alphanumeric and zero terminated. The ATMTSM will copy this name to a local structure. The maximum length is MAX_DRIVER_NAME.

Note Neither the card nor the driver names are length-preceded. They are regular C type strings.

board_name

The board name is an optional field that can be up to 17 characters long. It must be zero terminated.

driver_options

This field defines various driver options. For future compatibility, reserved bits must be set to zero. The following options are defined:

None (all bits are currently reserved and must be set to zero).

max_frame_size

This field contains the maximum frame size (in bytes) the driver is able to receive or send. The ATMTSM can return a lower value based on the maximum ECB size.

custom_stats_ptr

The *custom_stats_ptr* is a pointer to ATMTSM_CUSTOM_TEXT_LIST, which is an array of one or more pointers to custom text lines. Each line is zero byte terminated and no more than 42 characters long (excluding the zero byte). The array index (0..n) matches the custom counter number passed to *atmtsm_incr_custom_counter* or *atmtsm_update_custom_counter.* The last entry of the array must be a NULL pointer. Set this pointer to NULL when no HSM custom statistics are provided.

hsm_ads_size

If the driver wants the ATMTSM to allocate the HSM Adapter Data Space, the *hsm_ads_size* field contains the size of that ADS. A zero indicates that field *hsm_ads_handle* contains the HSM ADS handle.

hsm_shutdown_proc

If the driver wants to be called at shutdown time, it must insert its shutdown entry point here. NULL indicates no call. The driver does not need to call the ATMTSM with any deregistration because this procedure is typically called prior to a system shutdown that is similar to EXITing the server after DOWNing it.

hsm_ads_handle

If the *hsm_ads_size* field is non zero, the ATMTSM will return the HSM ADS handle. Otherwise the ATMTSM will use the given handle.

tsm_ads_handle

This handle is returned by the ATMTSM upon successful registration. It must be passed to the ATMTSM when the HSM passes control commands to the ATMTSM. It is also required when the HSM deregisters from the ATMTSM.

board_number

This value is returned by the ATMTSM upon successful completion of the registration. The HSM may need the board number to register with other applications (e.g. SNMP).

poll_support_level

The ATMTSM returns the *poll_support_level* value. The value is based on the platform on which the driver is loaded.

Poll Support Level	Definition
Zero	No polling support
One	Limited polling support
Two	Fair polling support
Three	Good polling support

Good polling support would be set when the polling rate is an inverse function of the load of the processor(s). The driver can use this value to determine whether it should use interrupts, polling or maybe a hybrid by default. The NetWare Server returns a 2. A DOS Client may return a 1, as its polling frequency usually depends on infrequent timer interrupts.

future_extension_n

Some room has been reserved for future expansion. The driver must initialize these fields to zeros. This allows for better backward compatibility.

Return values for atmtsm_register_hsm call

Upon completion of a call to *atmtsm_register_hsm*, the ATMTSM returns a zero value if no errors are detected. The following values are returned:

Return Value	Definition
0	HSM registration successful
-1	HSM and ATMTSM versions mismatch

Return Value	Definition
-2	Insufficient resources
-3	Could not register the board with LSL
-4	Firmware load failure

Deregistration

When the driver is being unloaded, it needs to deregister from the ATMTSM. Before deregistration, the driver must return all resources it has requested from the ATMTSM. For example, the driver must return all ECBs that it has allocated for receive queues. Any data that has been given to the HSM to be transmitted must also be returned.

Before deregistering, the driver must make sure that a STOP INTERFACE command has been given to the ATMTSM (assuming that a START INTERFACE was previously passed). The ATMTSM treats this as an implied request to disable all currently outstanding VCs. The driver may use the time between the STOP INTERFACE command and deregistration to do such things as cancel DMA transfers, stop interrupts, release queues, and so on. The driver should be aware that during deregistration all the assigned hardware resources will be released.

The ATMTSM will internally deregister all the registered hardware resources, thus the driver is not required to explicitly do this.

The driver deregisters itself from the ATMTSM by passing a DEREGISTER control command.

HSM Control Interfaces

After the driver has registered successfully, the ATMTSM indirectly passes control commands to the driver through the *hsm_control_proc* entry point. The driver must send control replies to some of the ATMTSM control commands. The control commands and replies deal mainly with enabling and disabling VCs. The ATMTSM passes commands to the HSM through the ATM_TSM_TO_HSM_CONTROL structure. The HSM must use the ATM_HSM_TO_TSM_CONTROL structure. Both the ATMTSM and HSM must reference these structures only during the time the command or reply is presented, thus, they can be dynamic (on the stack).

The ATMTSM calls the HSM with a control command through the control entry point passed during registration. The HSM must call the ATMTSM *atmtsm_hsm_control* procedure. In both cases, the only parameter is the address of the appropriate control structure.

Assignment of ATMTSM and HSM handles

To minimize the lookup of VC-related tables by both the HSM and ATMTSM, each assigns their own VC- specific handle. The assignment is done during enabling of the VC. Because the ATMTSM initiates the VC enable, it assigns the *tsm_vc_handle* first, and passes the value as a “pointer to void” to the HSM, when it presents the HSM with the control command to enable the VC. When the HSM replies, it must always pass that *tsm_vc_handle* back to the ATMTSM. The same handle must be passed when the HSM presents the ATMTSM with received frames for that VC.

The HSM can assign the *hsm_vc_handle*, and either pass it to the ATMTSM upon return from the command to enable the VC, or when it replies that the VC has been enabled. The latter is done for drivers that utilize a processor on the board to make the assignment, in which case that handle might not be known until the driver gets an acknowledgment from the firmware.

During contract verification, there will just be a *tsm_contract_handle*. The HSM must return this handle when replying with the CONTRACT VERIFIED.

Note In the case the driver returns the *hsm_vc_handle* during the reply to the ENABLE VC command, the ATMTSM may need to disable the VC before the driver has been able to report the *hsm_vc_handle* back to the ATMTSM. To indicate which VC must be disabled, the ATMTSM therefore also passes the VCI to the HSM. Such a driver must be able to locate the VC-related information based on the VCI.

The handles are “pointers to void”. The ATMTSM has no need to know what the HSM's handle is (it can be the address of a VC Control Block, an index into an array, or the VCI itself). Conversely, the HSM must never dereference the *tsm_vc_handle*.

ATMTSM to HSM Control Commands

The ATMTSM to HSM control commands are passed through the entry point *hsm_control_proc*. This entry point is passed to the ATMTSM during the HSM registration. The following control commands are defined:

ATM_TSM_TO_HSM_VERIFY_CONTRACT

This command is given to the HSM before the ATMTSM attempts to enable a VC. Its purpose is for the HSM to verify that its hardware is capable of performing the desired contract. The HSM must not reserve any bandwidth at this time, because there is no guarantee that the actual VC will be enabled. The HSM must reply with a CONTRACT VERIFIED. The HSM can call the ATMTSM with this reply before it returns to the ATMTSM from processing the VERIFY CONTRACT command.

ATM_TSM_TO_HSM_ENABLE_VC

This command is given to the HSM when the actual VC can be enabled. The HSM can assign the *hsm_vc_handle* now, and return it to the ATMTSM when it has completed the handling of this indication, or it can return the *hsm_vc_handle* when it passes the VC ENABLED reply to the ATMTSM. The HSM is allowed to pass the VC ENABLED reply while processing the ENABLE VC command, or can wait until its firmware has completely enabled the VC.

ATM_TSM_TO_HSM_DISABLE_VC

This command is given to the HSM when an earlier enabled VC needs to be disabled. If the HSM is still in the process of enabling the VC, and has not yet passed the *hsm_vc_handle*, the HSM must use the VCI. This command is unilateral. The HSM must not pass any more data or control indications for the disabled VC.

ATM_TSM_TO_HSM_CONTROL_CHANGE

This command is given to the HSM when the ATMTSM needs to change the receive control entry points. This can be done when a remote monitor application wants to begin or end the monitoring of a particular VC.

The ATMTSM to HSM control commands all have a number of fields in common. When required individual commands have one or more specific fields. As an example, the fields for the command VERIFY CONTRACT are referenced by qualifying the control structure name by “<name>.info.verify_contract”. The common fields are described first, followed by the various fields for the individual controls.

Common fields of the ATMTSM to HSM Control

control_type

The type of control command that is being given to the HSM.

hsm_ads_handle

The HSM ADS handle assigned after registration.

ATM_TSM_TO_HSM_VERIFY_CONTRACT

tsm_contract_handle

This handle is assigned by the ATMTSM. The HSM must return this handle when it responds to this indication.

qos

This structure contains the Quality of Service parameters the HSM is asked to verify. Refer to the Common Definitions section for more information.

ATM_TSM_TO_HSM_ENABLE_VC

tsm_vc_handle

This handle is assigned by the ATMTSM. This is the handle that the HSM must use when it responds to this indication, as well as when it receives frames for this VC.

vpi_vci

The Virtual Connection Identifier (VCI).

qos

This structure contains the Quality of Service parameters to be used for this VC. Refer to the Common Definitions section for more information.

hsm_vc_handle

This is a returned value. It is assigned by the HSM and optionally returned here. The ATMTSM assigns a NULL_ATM_HANDLE value here, so the HSM does not need to update this field when it returns the *hsm_vc_handle* later (when it replies to this indication with a VC ENABLED).

tsm_rcv_proc

The HSM must use this entry point to forward received data for the VC to the ATMTSM.

tsm_rcv_get_ecb_proc

The HSM must use this entry point to forward received data for the VC to the ATMTSM, when it also needs to obtain a new receive ECB.

ATM_TSM_TO_HSM_DISABLE_VC

vpi_vci

The VCI.

hsm_vc_handle

The HSM's handle.

Note This handle may not yet have been assigned by the HSM, in which case the HSM must use the VCI. The value will be NULL_ATM_HANDLE when unassigned.

ATM_TSM_TO_HSM_CONTROL_CHANGE

hsm_vc_handle

This is the HSM's handle for the VC.

tsm_rcv_proc

The HSM must use this entry point to forward received data for the VC to the ATMTSM.

tsm_rcv_get_ecb_proc

The HSM must use this entry point to forward received data for the VC to the ATMTSM, when it also needs to obtain a new receive ECB.

HSM to ATMTSM Control Commands and Replies

The following HSM to ATMTSM control commands and replies are defined (and they are passed to the ATMTSM by calling *atmtsm_hsm_control*):

ATM_HSM_TO_TSM_REGISTER_HW

This command allows the HSM to register its needed hardware resources. The HSM may register/deregister its hardware multiple times, as some hardware may be needed to determine the rest of the required hardware. The final hardware registration must contain all the information needed for normal operation.

ATM_HSM_TO_TSM_DEREGISTER_HW

This command allows the HSM to deregister the used hardware resources.

ATM_HSM_TO_TSM_DEREGISTER

This command is given by the HSM to deregister the interface. It must be the last control command for the given interface.

ATM_HSM_TO_TSM_START_INTERFACE

This command must be passed to the ATMTSM after successful registration. It signals to the ATMTSM that the driver is ready to start normal operations on the given interface (identified by the ATMTSM ADS handle). The ATMTSM does not attempt to enable any VCs (or request to verify any contracts), until the driver has given this indication.

ATM_HSM_TO_TSM_STOP_INTERFACE

This command must be passed to the ATMTSM before the driver deregisters. The ATMTSM internally releases all the enabled VCs without informing the HSM. The HSM can also use the STOP/START commands as a way to release all current VCs. This can be useful after certain hardware errors.

Note The HSM is allowed to have unanswered VERIFY CONTRACT or ENABLE VC commands outstanding. The ATMTSM internally releases those as well.

ATM_HSM_TO_TSM_CONTRACT_VERIFIED

The HSM must respond to a VERIFY CONTRACT request from the ATMTSM with this reply. A boolean accepted indicates whether the HSM can perform the given QoS parameters. Most QoS parameters deal with transmission rates. If the QoS specifies that the HSM can adjust the rates, and the HSM needs to do this, the HSM must report the changed rates.

ATM_HSM_TO_TSM_VC_ENABLED

The HSM must respond to an ENABLE VC request from the ATMTSM with this reply. The ATMTSM does not present the HSM with any data to send for this VC until this reply is received.

ATM_HSM_TO_TSM_DISABLE_VC

This command can be passed by the HSM when certain errors are detected for the given VC. It is unilateral. The ATMTSM must not present the HSM with any more data or control indications (for the VC) after this command is received.

The HSM may send this command as a reply to an ENABLE VC command.

ATM_HSM_TO_TSM_STOP_OUTPUT

When the HSM detects that the send queue for a VC is getting full, it can pass this command to the ATMTSM. The ATMTSM ceases passing send indications to the HSM for the given VC. The HSM must pass the START OUTPUT command before the ATMTSM continues sending data.

ATM_HSM_TO_TSM_START_OUTPUT

This command must be sent to the ATMTSM after the send queue for the given VC has sufficient room for more output.

The HSM to the ATMTSM control commands and replies all have a number of fields in common. When required, individual indications have one or more specific fields. As an example, the individual fields for the reply CONTRACT VERIFIED are referenced by qualifying the control structure name with

“<name>.info.contract_verified”. The common fields are described first, followed by the various fields for the individual controls.

Common fields of the HSM to ATMTSM Control

control_type

The type of control command or reply that is being passed to the ATMTSM.

tsm_ads_handle

The ATMTSM ADS handle, as assigned during registration.

ATM_HSM_TO_TSM_REGISTER_HW

The hardware specific fields are passed through a pointer to structure ATM_HSM_HW_INFO. This structure contains the following fields.

transmission_type

This field is a pointer to a string that contains a description of the transmission type (such as “4B5B”, “SDH”, “IBM”). The maximum length is MAX_TRANSMISSION_TYPE. This information is used for network management purposes only.

media_type

This field is a pointer to a string that contains a description of the media type (such as “COAX”, “UTPx”, “FIBER”). The maximum length is MAX_MEDIA_TYPE. This information is used for network management purposes only.

speed

This field refers to the speed of the interface. It can be specified in either Kilobits per second, Megabits per second or Gigabits per second, based on the speed_base.

speed_base

This field identifies whether the speed is in Kilobits per second (SPEED_BASE_KBPS), Megabits per second (SPEED_BASE_MBPS) or Gigabits per second (SPEED_BASE_GBPS). A speed of 155.52 Megabits per second can be passed as 155 Megabits per second, or more accurately as 155,520 Kilobits per second.

max_cell_rate

This field refers to the maximum cell rate (per second). For a 155 Megabits per second Sonet interface with a payload capacity of 149.76 Megabits per second, this value will be 353,207 (149,760,000 / (53 * 8).

upper_vpc

This field contains the upper value for the Virtual Path Connection number the HSM supports (0..255). The current implementation of the ATMTSM assumes a value of zero.

upper_vcc

This field contains the upper value for the Virtual Channel Connection number the HSM supports (n..65535). A minimum value of 32 allows for a single SVC.

max_physical_fragments

When the driver uses *atmtsm_logical_to_physical_list*, it must specify the maximum number of physical fragments available for that conversion. Note that a single logical fragment may result in multiple physical fragments because the logical fragment can cross physical page boundaries. Since the maximum number of logical fragments is 16, the theoretical maximum for this value is 47, providing the maximum frame size is 64 KBytes and the system has 4 KByte pages.

Note A single logical fragment may result in multiple physical fragments, as the logical fragment can cross physical page boundaries.

esi

This field contains the 6-byte End Station Identifier (ESI). The HSM may obtain the ESI through board ROM, or other configuration options. The ATMTSM will use the ESI to form the default ATM Address for this interface.

hsm_isr_proc

This defines the entry point that receives the interrupts. The entry point must be of C language type ATMHSM_ISR_PROC.

Set to NULL if not required.

sharing_flags

These flags indicate whether the driver is sharing some of the resources with another interface. Resources such as Interrupt, DMA, Port, and Memory can be shared. Set to zero if none are being shared.

bus_tag

This field is for the platform dependent tag of the bus. The HSM may obtain the bus tag by calling *CMSMGetBusTag.* Set *bus_tag* to NULL if it is not being used or is unknown.

slot_number

This field refers to the slot number and should be set to zero if not used.

channel_number

This field is used for multichannel adapters. Set to zero to indicate that none is used.

port0

This field contains the first IO port number and should be set to zero if not used.

port0_range

This field specifies how many port numbers are associated with port0 and should be set to zero if not used.

port1

This field contains the second IO port number and should be set to zero if not used.

port1_range

This field specifies how many port numbers are associated with port1 and should be set to zero if not used.

memory0_address

This field contains the first shared memory address and should be set to zero if not used.

memory0_bytes

This field specifies the number of bytes starting at memory0_address and should be set to zero if not used.

linear0_address

The ATMTSM returns this value. It is the linear (logical) address for the first shared memory area.

memory1_address

This field contains the second shared memory address and should be set to zero if not used.

memory1_bytes

This field specifies the number of bytes used starting at memory1_address and should be set to zero if not used. The same limitation as for memory0_bytes applies.

linear1_address

The ATMTSM returns this value. It is the linear (logical) address for the second shared memory area.

interrupt_number

This field defines the interrupt request level and should be set to 0xff when not used.

dma_line

This field defines the DMA channel number and should be set to 0xff when not used.

ATM_HSM_TO_TSM_DEREGISTER_HW

No additional fields are defined.

ATM_HSM_TO_TSM_DEREGISTER

No additional fields are defined.

ATM_HSM_TO_TSM_START_INTERFACE

No additional fields are defined.

ATM_HSM_TO_TSM_STOP_INTERFACE

No additional fields are defined.

ATM_HSM_TO_TSM_CONTRACT_VERIFIED

tsm_contract_handle

This handle must be the same as passed by the ATMTSM when it requested the HSM to perform the contract verification. After the CONTRACT VERIFIED reply, this handle is obsolete.

qos

This structure contains the Quality of Service parameters the HSM is able to support. It must be the same or as close to the same requested by the ATMTSM. The QoS information is passed completely but the HSM should only be reading or using fields that make sense. A backward QoS is also used to allow smart drivers to fine tune their receivers. Refer to the Common Definitions section for more information.

return_code

This return field is set to zero when the HSM accepts the given QoS for the contract. Values are HSM_CONTRACT_ACCEPTED and HSM_CONTRACT_CANNOT_DO_RATE.

ATM_HSM_TO_TSM_VC_ENABLED

tsm_vc_handle

This handle must be the same as passed by the ATMTSM when it requested the HSM to enable the VC.

hsm_vc_handle

This handle is returned by the HSM. The HSM might already have returned this handle when it was called by the ATMTSM's ENABLE VC command.

ATM_HSM_TO_TSM_DISABLE_VC

tsm_vc_handle

The ATMTSM-assigned handle for the VC.

reason_code

The reason why the HSM disabled the VC.

ATM_HSM_TO_TSM_STOP_OUTPUT

tsm_vc_handle

The ATMTSM-assigned handle for the VC.

ATM_HSM_TO_TSM_START_OUTPUT

tsm_vc_handle

The ATMTSM-assigned handle for the VC.

Return values for "atmtsm_hsm_control" call

Upon completion of a call to *atmtsm_hsm_control*, the ATMTSM returns a zero value when no errors are detected. When errors are detected, the ATMTSM will have logged appropriate warning messages on the console. The following values are returned:

Return Value	Definition
0	Control completed successfully
-1	Invalid ATMTSM ADS handle
-2	Invalid control type
-3	Hardware registration failed
-4	START/STOP INTERFACE out of order
-5	Could not start Signaling during START INTERFACE
-6	Interface must be STOPPED first
-7	Invalid ATMTSM VC handle

Get/Return Buffers (ECBs)

Event Control Blocks (ECBs) are NetWare buffers. The HSM might require ECBs to prepare the receive queue(s). It can call *atmtsm_get_ecb* to obtain an ECB and *atmtsm_return_ecb* to release it (typically at deregistration time). When the driver forwards received data and needs another ECB, it can combine the two above mentioned procedures by indirectly calling the procedure *tsm_rcv_get_ecb_proc*. Transmit buffers must never be returned through *atmtsm_return_ecb*, because the driver cannot be sure that the applications actually use ECBs.

Both *atmtsm_get_ecb* and *tsm_rcv_get_ecb_proc* calls will insert the logical address for the receive data (start-address). Since receive ECBs will be guaranteed to have a physically contiguous receive area, the ATMTSM will also insert the physical receive data start-address. A macro called PHYSICAL_RECV_ADDRESS is provided for HSMs to extract the physical start-address from the ECB. This eliminates the need for an additional procedure call.

Data Interfaces

The data interfaces between the HSM and the ATMTSM are available for a particular VC, after the VC has been enabled. After either the ATMTSM or HSM has disabled the VC, these interfaces cannot be used anymore. Any obsolete *tsm_vc_handle* used by the driver might ABEND the system. However, the ATMTSM can prevent this for a short time after the VC has been released, because it does not immediately release the control block associated with the *tsm_vc_handle*.

The data interface between the ATMTSM and HSM is “ECB Aware” in both directions. When using AAL5, the HSM must be able to forward the two padding bytes (called CPCS/U) of the last cell of each frame. Currently all VCs should be AAL5. The macro CPCS_U allows the HSM to store or extract these bytes from the ECB.

The macro RECEIVE_FLAGS is available for the HSM to report the recognition of congestion to the User Application.

Receive

After the HSM has completed the reception of a frame for an enabled VC, it must indirectly call *tsm_rcv_proc*. If the HSM wants to forward received data and obtain a new ECB, it can indirectly call *rcv_get_ecb_proc*. The latter is the same as calling procedure *tsm_rcv_proc* (indirectly) followed by *atmtsm_get_ecb* (directly). Both procedures require the address of the receive ECB and the *tsm_vc_handle* as parameters, while *tsm_rcv_get_ecb_proc* will return a new receive ECB (this new ECB has no association with the VC). When the HSM obtains a receive ECB, the ATMTSM will insert the address where the HSM must start copying the frame to. This address will be in field

ECB_Fragment[0].FragmentAddress. The frame must be copied as a single fragment.

The ATMTSM will leave at least the size of a pointer of space below the starting address of the data.

The HSM can use this as a place to store the ECB address, when its DMA only returns the data address. The HSM does not need to fill out any fields in the ECB other than ECB_Fragment[0].FragmentLength and ECB_DataLength. The ECB_FragmentCount is set to 1 when the ECB is acquired. The HSM can use the fields ECB_NextLink and ECB_PreviousLink for queuing or other purposes. Because the HSM passes the ECB to the ATMTSM, accompanied by the *tsm_vc_handle*, the HSM does not even need to fill in the ECB_BoardNumber.

Transmit

The ATMTSM passes the frame to be transmitted to the HSM by calling the ATM HSM entry point *hsm_send_proc*. This entry point is passed during registration of the HSM. The ATMTSM passes the ECB address and the *hsm_vc_handle*. After the HSM has sent the frame, it must call *atmtsm_send_complete*. This procedure requires just the ECB address as a parameter. HSM must accept fragmented transmit ECBs. There will never be more than 16 fragments. HSM can use procedure *atmtsm_logical_to_physical_list* to convert the logical memory address to a physical one. HSM can use the fields *ECB_NextLink* and *ECB_PreviousLink* for queuing. It can also use field *ECB_DriverWorkspace*. Other fields in the ECB must not be modified by HSM. As for Receive, the *ECB_BoardNumber* will not be set by the ATMTSM.

Interrupts

When applicable, HSM can pass an Interrupt Service Routine (ISR) address to the ATMTSM during registration. When an interrupt occurs, the ATMTSM will call HSM at that address. The only additional parameter passed to HSM is *hsm_ads_handle*. HSM must report the *hsm_ads_handle* to the ATMTSM before it expects the first interrupt. The control command that allows HSM to do this is either PASS ADS HANDLE or START INTERFACE. The HSM can also assign unique ISR addresses for each individual interface, in which case interrupts are allowed prior to passing the *hsm_ads_handle*.

Polling / Periodic Calls

After registration, HSM can request the ATMTSM to be polled regularly. This is done by passing the poll entry point address to *atmtsm_poll_address*. Polling is done on a board by board basis. HSM can stop the polling by passing a NULL poll entry point address. HSM can change the poll entry point address at any time.

The driver can also ask the ATMTSM for periodic calls. The HSM must call *atmtsm_periodic_call_address*, passing the entry point to be called, as well as the time between calls. Again, passing a NULL entry point will stop the periodic calls. The given time is in milliseconds, however the ATMTSM will round it up to the nearest tick interval. On PC platforms, the tick interval is 55 Milliseconds.

The HSM is called back at process level.

Statistics

The ATMTSM supports General Statistics as well as Custom Statistics. The custom statistics are divided in two groups. One group is defined and maintained by the ATMTSM and one group is (optionally) defined and maintained by the HSM. During registration, the HSM passes an array of pointers to its custom statistics text lines, terminated by a NULL pointer after the last one. Three procedures are provided for the HSM to increment or update its custom statistic counters. These procedures are:

atmtsm_incr_custom_counter

atmtsm_update_custom_counter

atmtsm_set_custom_counter

Additional ATMTSM Interfaces and Interface Summary

This section describes the above defined and some additional interfaces that are provided by the ATMTSM for use by the HSM. All interfaces are provided through direct calls.

atmtsm_register_hsm

The HSM registers with the ATMTSM after successful load.

The C definition is:

long atmtsm_register_hsm (ATMTSM_REGISTER_HSM_CONTROL*);

atmtsm_hsm_control

The HSM can call this procedure to pass control information to the ATMTSM.

The C definition is:

long atmtsm_hsm_control (ATM_HSM_TO_TSM_CONTROL*);

atmtsm_poll_address

The HSM can call this procedure to pass its poll address to the ATMTSM. When a NULL address is passed, the ATMTSM will cease calling the HSM. A single parameter *hsm_ads_handle* is passed to the HSM with every call to the poll procedure.

Note This procedure is called individually for each interface (board).

The C definition is:

```
void atmtsm_poll_address (ATM_HANDLE tsm_ads_handle,
    ATMHSM_POLL_PROC* hsm_poll_proc,
    UINT32 minimum_microseconds_delay);
```

atmtsm_periodic_call_address

The HSM can call this procedure to pass its periodic call address to the ATMTSM. When a NULL address is passed, the ATMTSM will cease calling the HSM. A single parameter *hsm_ads_handle* is passed to the HSM with every periodic call.

Note This procedure is called individually for each interface (board).

The C definition is:

```
void atmtsm_periodic_call_address (ATM_HANDLE tsm_ads_handle,
    ATMHSM_PERIODIC_CALL_PROC* hsm_call_proc,
    UINT32 milliseconds);
```

atmtsm_micro_delay

This procedure is available for the HSMs to introduce a small processor delay. The given value can only be between 0 and 4095 microseconds. This procedure also returns the current value of the high resolution timer. It can be called with interrupts enabled or disabled.

The C definition is:

UINT32 atmtsm_micro_delay (UINT32 micro_seconds);

atmtsm_get_ecb

This procedure can be used when the HSM needs ECBs for the receive queue(s). A NULL is returned when no ECB is available at the moment.

The C definition is:

struct ATMECB* atmtsm_get_ecb (ATM_HANDLE tsm_ads_handle);

atmtsm_return_ecb

The HSM calls this procedure to return unused receive ECBs.

The C definition is:

void atmtsm_return_ecb (struct ATMECB* ecb_ptr);

atmtsm_send_complete

After transmission of a frame (or when the HSM is being unloaded), the driver must return the transmission resources by calling this procedure.

The C definition is:

void atmtsm_send_complete (struct ATMECB* ecb_ptr);

atmtsm_printf

This procedure allows the HSM to print (error) messages on the console. It allows for National Language Enabled messages. The maximum number of parameters is 8 (excluding the format string). This procedure can only be called during driver LOAD and UNLOAD.

The C definitions is:

int atmtsm_printf (const char* format, ...);

atmtsm_alloc

This procedure allocate memory for the HSM. The HSM must pass the *tsm_ads_handle* as well as the number of bytes. If at this moment not enough memory is available, a NULL will be returned.

The C definitions is:

void* atmtsm_alloc (ATM_HANDLE tsm_ads_handle, UINT32 bytes);

atmtsm_free

This procedure free the memory allocated through *atmtsm_alloc*.

The C definitions is:

void atmtsm_free (void* alloc_ptr);

atmtsm_logical_to_physical_list

This procedure allows the HSM to convert the given list of logical memory addresses to a list of physical memory addresses. Each list consists of one or more fragment descriptors. Each fragment descriptor consists of an address and a size. The return value is the number of physical fragments generated (a zero indicates an error, and the HSM must discard the associated data). The maximum size of the HSM physical list is passed during registration.

The C definition for *atmtsm_logical_to_physical_list* is:

```
UINT32 atmtsm_logical_to_physical_list

    (struct atm_ads_rec* tsm_ads_handle,

    INT8 logical_fragments,

    ATM_FRAGMENT* logical_list,

    ATM_FRAGMENT* physical_list);
```

atmtsm_logical_to_physical

This procedure converts the given logical memory address to the physical memory address. This procedure can only be used for allocated memory and receive ECBs. It cannot be used for shared RAM. If needed, the driver must remember the difference between the logical and physical shared RAM addresses, and use it to do the conversion locally. For transmit ECB fragments, the driver must use *atmtsm_logical_to_physical_list*.

The C definition is:

```
void* atmtsm_logical_to_physical (void* logical_address);
```

atmtsm_physical_to_logical

This procedure converts the given physical memory address to the logical memory address.

The C definition is:

```
void* atmtsm_physical_to_logical (void* physical_address);
```

atmtsm_get_firmware

The procedure provides the HSM with the address and length of its firmware. The HSM can only call this procedure while the interface is stopped. As soon as the HSM passes the START INTERFACE command, the information is obsolete.

The C definition is:

```
UINT8* atmtsm_get_firmware (ATM_HANDLE tsm_ads_handle,
    UINT32* firmware_length);
```

atmtsm_rcv_oam_cell

The procedure is called by the HSM to pass incoming OAM cells for previously established VCs to the ATMTSM.

The C definition is:

```
void atmtsm_rcv_oam_cell (ATM_HANDLE tsm_vc_handle,
    struct ATMECB* ecb_ptr);
```

atmtsm_rcv_vc_error

This procedure is called by the HSM to report an error, recognized during reception of a frame for the given VC. If an ECB was used, it is also passed (and released). Each error type has an associated ATMTSM custom statistic counter, which the ATMTSM will increment.

The C definition is:

```
void atmtsm_rcv_vc_error (ATM_HANDLE tsm_vc_handle,
    struct ATMECB* ecb_ptr, UINT8 error_type);
```

atmtsm_rcv_error

This procedure is called by the HSM to report an error, recognized during reception of a frame (not associated with an enabled VC). If an ECB was used, it is also passed (and released). Each error type has an associated ATMTSM custom statistic counter, which ATMTSM will increment.

The C definition is:

```
void atmtsm_rcv_error (ATM_HANDLE tsm_ads_handle,
    struct ATMECB* ecb_ptr, UINT8 error_type);
```

atmtsm_incr_custom_counter

This procedure is called by the HSM when it wants to increment the given HSM custom counter.

The *counter_number* must match the relative index (0..n) in the array of HSM custom counter text lines, passed during registration of the HSM.

The C definition is:

```
void atmtsm_incr_custom_counter (ATM_HANDLE tsm_ads_handle,
    UINT8 custom_counter_number);
```

atmtsm_update_custom_counter

This procedure is functionally the same as *atmtsm_incr_custom_counter*, with the exception that the HSM passes the (signed) value to update the counter with.

The C definition is:

void atmtsm_update_custom_counter (ATM_HANDLE tsm_ads_handle,

UINT8 custom_counter_number, long update_value);

atmtsm_set_custom_counter

This procedure set the given custom counter to the given value.

The C definition is:

void atmtsm_set_custom_counter (ATM_HANDLE tsm_ads_handle,

UINT8 custom_counter_number, UINT32 value);

atmtsm_register_lock

This procedure must be called by the HSM to register and initialize a lock structure. The HSM can have as many locks as it requires (e.g. per interface, per receiver/transmitter, per VC, etc.). A lock can be reused many times, and must be deregistered prior to the HSM unload. This procedure returns a TRUE if the lock is successfully registered initialized. The *lock_id* is for debugging purposes, and must point to a short (statically defined) string.

The C definition is:

UINT8 atmtsm_register_lock (ATMLOCK* lock_ptr, char* lock_id);

atmtsm_deregister_lock

When the HSM is done using the lock, it must call the following procedure.

The C definition is:

void atmtsm_deregister_lock (ATMLOCK* lock_ptr);

atmtsm_lock / atmtsm_lock_ni

These procedures will obtain the lock. The first one will also disable interrupts. The second one assumes the HSM has disabled them. Both procedures spin to obtain the lock, when multiple processors are found. The system will abort when the lock cannot be obtained within a few seconds.

The C definition is:

```
void atmtsm_lock (ATMLOCK* lock_ptr);

void atmtsm_lock_ni (ATMLOCK* lock_ptr);
```

atmtsm_conditional_lock / atmtsm_conditional_lock_ni

These procedures will obtain the lock conditionally. If the lock cannot be obtained on the first attempt, a FALSE will be returned. In a single processor environment, the lock will always be obtained.

The C definition is:

```
UINT8 atmtsm_conditional_lock (ATMLOCK* lock_ptr);

UINT8 atmtsm_conditional_lock_ni (ATMLOCK* lock_ptr);
```

atmtsm_unlock / atmtsm_unlock_ni

These procedures will release the lock. When the lock was obtained with a *atmtsm_lock*, it must be released with *atmtsm_unlock*. The same is true for *atmtsm_lock_ni* and *atmtsm_unlock_ni*.

The C definition is:

```
void atmtsm_unlock (ATMLOCK* lock_ptr);

void atmtsm_unlock_ni (ATMLOCK* lock_ptr);
```

tsm_rcv_proc / tsm_rcv_get_ecb_proc (indirect)

The HSM calls one of these procedure indirectly to forward received data to the ATMTSM. The ATMTSM will provide this procedure address with the ENABLE VC indication. The reason these procedures are called indirectly is to allow the ATMTSM defined monitoring procedures. Those procedures can be changed on a per VC basis. Thus the HSM must retain these procedure addresses separate for each VC.

The C definitions are:

```
typedef void ATMHSM_TSM_RCV_PROC

    (ATM_HANDLE tsm_vc_handle, struct ATMECB* ecb_ptr);

typedef struct ATMECB* ATMHSM_TSM_RCV_GET_ECB_PROC

    (ATM_HANDLE tsm_vc_handle, struct ATMECB* ecb_ptr);
```

HSM provided interfaces

The following are the procedure entry-points called by the ATMTSM. All entry-points are called indirectly, and are mostly passed to the ATMTSM during the HSM registration.

hsm_control_proc (indirect)

The ATMTSM calls this procedure indirectly to pass control information to the HSM.

The C definition is:

```
typedef void ATMTSM_HSM_CONTROL_PROC

    (ATM_TSM_TO_HSM_CONTROL*);
```

hsm_send_proc (indirect)

The ATMTSM calls this procedure indirectly to pass data to be transmitted to the HSM.

The C definition is:

typedef void ATMTSM_HSM_SEND_PROC

(ATM_HANDLE hsm_vc_handle, struct ATMECB* ecb_ptr);

hsm_isr_proc (indirect)

The ATMTSM calls this procedure indirectly to pass and interrupt event to the HSM.

The C definition is:

typedef void ATMHSM_ISR_PROC (ATM_HANDLE hsm_ads_handle);

hsm_poll_proc (indirect)

The ATMTSM calls this procedure indirectly. This procedure address is passed to the ATMTSM through a call to *atmtsm_poll_address*.

The C definition is:

typedef void ATMHSM_POLL_PROC (ATM_HANDLE hsm_ads_handle);

hsm_periodic_call_proc (indirect)

The ATMTSM calls this procedure indirectly and periodically. This procedure address is passed to the ATMTSM through a call to *atmtsm_periodic_call_address*.

The C definition is:

typedef void ATMHSM_PERIODIC_CALL_PROC

(ATM_HANDLE hsm_ads_handle);

Performance issues and the HSM design

Since ATM can deliver a high volume of data it is important that performance related issues are addressed during the design of the HSM. This section discusses the problems and provides some tips on how to prevent the high speed ATM interface(s) from congesting the system's CPU.

Receive

Experience in the past has shown that the CPU performance kept pace with, or outperformed available LAN capacity. However, multiple Ethernet or Token-Ring networks have been able to choke 486 systems. This has been particularly true for higher performance LAN drivers that forward all received frames at an interrupt priority level. If such a driver does not do any input flow control, it will simply fill all the receive ECBs. Those ECBs then sit in a Protocol Stacks' receive queues, waiting for the interrupt level processing to be completed. Since the interrupt level processing time roughly increases linearly with the offered load, there is not enough time left for the Protocol Stacks to handle the received frames. Thus the various receive queues just fill up until the system runs out of buffers. Once that happens, some drivers still keep the interrupt level logic busy with the frames to be discarded. And the system has come to a grinding halt.

One sometimes proposed solution has been to not just pass the received frames at interrupt level, but also forward the frames at the interrupt level. Thus running close to 100% of the CPU cycles at an interrupt level. This kind of solution limits the usefulness of interrupts, and interfaces that really require interrupts (clocks, error handling, etc.) are losing them due to overruns.

Of course, low performance LAN drivers by design do not cause any such problems, as they drop incoming frames before they are able to saturate the system's CPU. The trick for a driver is to find out when (in combination with other interface drivers) it is making the CPU congested, and only then to start dropping incoming frames.

One method to accomplish this is for the driver to use a polling mechanism, in conjunction with a limited receive queue (limited means less than all the available ECBs). Once the CPU becomes congested, the ATM driver can fill the limited receive ECB queue. There is no problem if the driver is polled before the queue is completely filled However, if the queue is full, and more frames arrive, the driver could use the Quality of Service defined priority. There is a priority assigned to every Virtual Connection. Lower priority frames are more likely candidates to be dropped, while higher priority frames could even use the receive ECBs that are currently occupied with lower priority frames. If the CPU remains congested, the driver eventually starts to drop the high priority frames.

The above scheme is adaptive and fair. It also gives the driver a means to favor higher priority data over lower. It would be much more complex to do something like this with interrupts. One important thing to remember is that the HSM should spend as little CPU time as possible on frames it is going to discard. If a lot of time is being spent on discarded frames, some of the above described behavior might re-appear.

Transmit

The transmission related issues are not as critical as the ones for receive. The main reason is that transmission of end station data can never outpace the available CPU, as the CPU will just limit the amount to what it can generate. For data that has been received from another interface and that interface does a good job of flow controlling the input (as suggested in the previous section), problems will likely be minimal. It is however important that the driver will give sufficient priority to the output logic, otherwise all output may end up in the transmit queues.

Transmit versus Receive Priority

Intermediate stations (routers and bridges) work best when the transmit logic is given some priority over receive. End stations (servers and clients) may work better when receive is given some priority over transmit. The latter because end stations have the ability to hold back additional output. Since the ATM HSM does not know whether it is servicing an end station versus an intermediate station, and quite often a NetWare Server is both, giving the receive and transmit equal priority is a good compromise.

Configurable Receive Modes

Unfortunately, the polling logic on client stations than run DOS, OS/2 or WINDOWS may not work very well, because applications are not required to yield. The minimum polling rate may depend on the system timer interrupt, which at 18 times per second is much too low. Polling in those environments may either be initiated by the HSM, through interrupts at rates of about once per Millisecond. If the HSM is not able to do this, an alternative is to use forwarding interrupts.

To make the same HSM work well in the many different environments, it is suggested that LOAD <driver-name> parameters are provided, that allow an HSM to operate in either a polled or interrupt mode. Possibly a hybrid solution of a poll interrupt when the receive queue grows above a certain threshold could be added as another configurable mode.

Novell®
Everything's Connected™

chapter 4 ATM HSM Definitions

Introduction

This section describes the definitions that are common to both the HSM interface and the User Application Interface.

Quality of Service Definition

The Quality of Service structure used for both HSM and the User Application interface. The QoS information is passed completely but the HSM should only be reading or using fields that make sense. It contains the following fields:

rate_type

The rate type defines whether the traffic cell rate is constant, variable (real-time or non real-time), available or unspecified. These definitions are defined: RATE_TYPE_CBR, RATE_TYPE_VBR_RT, RATE_TYPE_VCB_NON_RT, RATE_TYPE_ABR or RATE_TYPE_UBR (best effort).

rate_adjustment

This indicates to HSM whether and how to adjust the requested transmit rate(s). This should only be done when HSM cannot support the requested rate. Values are RATE_ADJUST_NONE, RATE_ADJUST_NEAREST, RATE_ADJUST_LOWER and RATE_ADJUST_HIGHER. HSM is only given the option to change the rate during the VERIFY CONTRACT phase, and must report the rate it can support during the CONTRACT VERIFIED response. If the request is for an EXACT rate and HSM cannot do that rate, it must report during the CONTRACT VERIFIED that it does not accept the contract.

priority

The priority assigned to this VC. The range is from 0 to 7. It is mainly for use by HSM.

direction

This is a two-dimensional array. The first element contains the forward and the second the backward traffic and other direction related parameters. If the user wants to ADD LEAFs later, the backward rates must be zero. A backward QoS is also used to allow smart drivers to fine tune their receivers. See below for further definition of its contents.

The direction array can be indexed with the constants ATM_FORWARD and ATM_BACKWARD. The direction is based on the User Applications point of view, such that the user does not need to check which end originated the call. The ATMTSM will perform the necessary inversion, because signaling assumes the forward is from the calling to the called party, and backward from the called to the calling party. Each of the two direction array elements contains the following fields (note that some fields only apply to a given rate type):

peak_cell_rate

This field defines the peak traffic rate in Cells per Second.

max_sdu_size

This field is the Maximum Service Data Unit size. It is only used for SVCs. A zero value indicates an unspecified size.

cell_loss_ratio

This field contains the Cell Loss Ration value (from 0 to 4). If the user has selected UBR, this field is ignored (and 0 is assumed).

sustained_cell_rate

This field defines the sustained traffic rate in Cells per Second. It is only used for VBR.

burst_size

This field defines the burst size for VBR.

cell_transfer_delay

This field defines the Cell Transfer Delay Tolerance. It is only used for CBR and VBR.

cell_delay_variation

This field defines the Cell Delay Variation Tolerance. It is only used for Real-time VBR.

Novell®
Everything's Connected™

chapter **5**

Virtual Connection Establishment Flow Diagrams

Introduction

Flow diagrams for Virtual Connection establishment This section shows message exchanges for establishing the various kinds of Virtual Connections.

The shown exchanges are between the User Application, ATMTSM, HSM and the ATM Switch. For completeness, some of the signaling protocol elements are shown. They are Call Setup, Call Proceeding, Connect and Connect Ack.

The User Application may start sending data as soon as the VC ESTABLISHED indication for either the PVC or SVC is received that has the *start_output* indication set to TRUE. The called end station will not start sending any data, until it has received the Connect Ack signaling protocol element. The ATMTSM will not forward any data to the User, until it has passed the VC ESTABLISHED indication to the User Application.

API sits at the bottom of the ATMTSM, while the User Application API sits at the top. An overview diagram is shown below.

Figure 5-1
User Application Virtual Connection

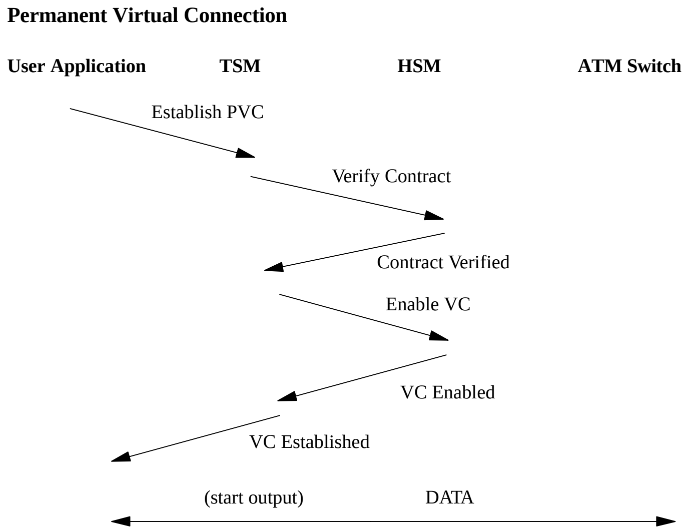


Figure 5-2
Outgoing Switched Virtual Connection

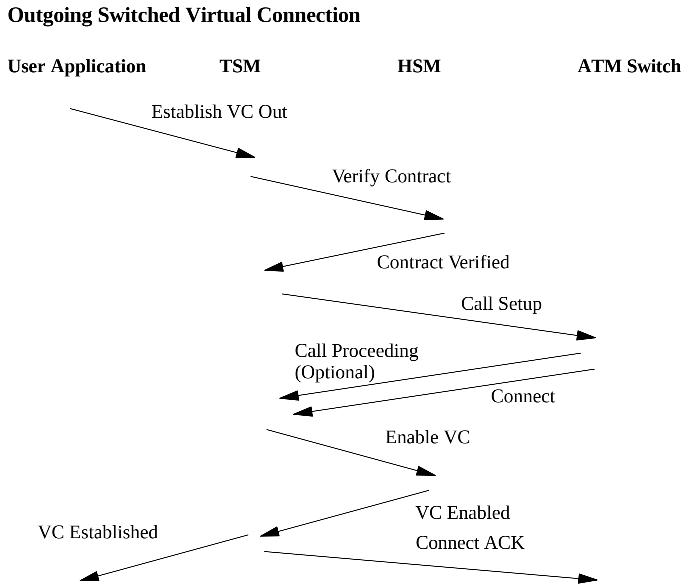
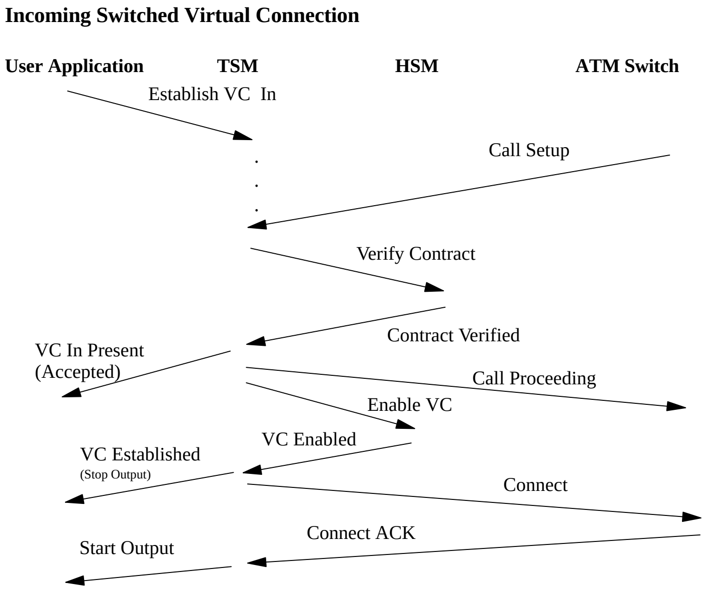


Figure 5-3
Incoming Switched Virtual Connection



Novell®
Everything's Connected™

chapter 6 Include Files

Introduction

This section lists the ATM HSM C Include files. Newer versions of these files may be included in soft copy format. Always use the latest version of these files.

- ATMQOS.H
- ATMHSMIF.H
- ATMHSM.IMP
- PARSER.H
- PARSER.IMP

ATMQOS.H

```
/*
 * (C) Copyright 1994-1996 Novell, Inc.  All Rights Reserved.
 *
 * File Name       : atmqos.h
 * Author          : Henk J. Bots
 * Created On      : Tue, Mar  8 (14:49:46) 1994
 * Last Modified By: Henk J. Bots
 * Last Modified On: Thu, Nov  2 (13:33:44) 1995
 * Update Count    : 48
 * Status          : Quality of Service record definitions
 */

/* -----------------------Begin Module Header ------------------------------

$Header:   K:/proj/atm/include/vcs/atmqos.h_v   1.1   02 Nov 1995 13:45:40   HBOTS  $

$Project: Novell MPR $

---------------------------------------------------------------------------
```

```
   This program is an unpublished copyrighted work which is proprietary to
   Novell, Inc.   and contains confidential information  that is not to be
   reproduced or disclosed  to any  other person  or entity  without prior
   written consent from Novell, Inc.  in each and every instance.

   WARNING:  Unauthorized  reproduction  of  this   program  as  well  as
   unauthorized  preparation of derivative works based upon the program or
   distribution of copies by sale, rental, lease or lending are violations
   of federal  copyright laws and  state trade secret  laws, punishable by
   civil and criminal penalties.

-----------------------------------------------------------------------------

$Log:   K:/proj/atm/include/vcs/atmqos.h_v  $
 *
 *    Rev 1.1   02 Nov 1995 13:45:40   HBOTS
 * Update Copyright message to include 1996.
 *
 *    Rev 1.0   21 Apr 1995 12:36:30   HBOTS
 * Correct a signaling problem with AAL parameters.

$EndLog: $

--------------------------- End Module Header --------------------------- */

#ifndef ATMQOS_H
#define ATMQOS_H

#ifndef UINT8
#define UINT8   unsigned char
#define UINT16  unsigned short
#define UINT32  unsigned long
#define INT8    signed char
#define INT16   signed short
#define INT32   signed long
#endif

#define NULL    0

#ifndef BYTE
#define BYTE    unsigned char
#define WORD    unsigned short
#define LONG    unsigned long
#endif

#ifndef ATM_HANDLE
#define ATM_HANDLE      void*
#define NULL_ATM_HANDLE ((ATM_HANDLE) -1)
#define no_additional_info int ignored
#endif

#define BYTES_PER_CELL  48
```

```
#define BITS_PER_CELL   (BYTES_PER_CELL * 8)

/*
 * Rate Type and Rate Adjustment definitions
 * ----------------------------------------
 */

#define RATE_TYPE_CBR           0   /* Constant */
#define RATE_TYPE_VBR_RT        1   /* Variable Real Time */
#define RATE_TYPE_VBR_NON_RT    2   /* Variable Non Real Time */
#define RATE_TYPE_ABR           3   /* Available */
#define RATE_TYPE_UBR           4   /* Unspecified */

#define RATE_ADJUST_NONE    0   /* No adjustment allowed */
#define RATE_ADJUST_NEAREST 1   /* Adjust to nearest (higher or lower) */
#define RATE_ADJUST_LOWER   2   /* Adjust to nearest lower value */
#define RATE_ADJUST_HIGHER  3   /* Adjust to nearest higher value */

#define ADJUSTED_NEAREST_CELL_RATE( bps_rate )     \
    ((bps_rate + BITS_PER_CELL/2) / BITS_PER_CELL)
#define ADJUSTED_LOWER_CELL_RATE( bps_rate )       \
    (bps_rate / BITS_PER_CELL)
#define ADJUSTED_HIGHER_CELL_RATE( bps_rate )      \
    ((bps_rate + BITS_PER_CELL-1) / BITS_PER_CELL)

/*
 * Cell Loss Ratio definitions
 * ---------------------------
 *
 * Current switches support only "undefined".  For UBR, undefined must
 * be used.
 */

#define CLR_UNDEFINED       0
#define CLR_HIGH_LOSS       1   /* > 10**(-5) */
#define CLR_MEDIUM_LOSS     2   /*   10**(-5) range */
#define CLR_LOW_LOSS        3   /*   10**(-7) range */
#define CLR_VERY_LOW_LOSS   4   /* < 10**(-7) */

/*
 * Note that forward is always the outgoing, and backward the incoming
 * direction,  regardless of how the connections was established.  The
 * HSM may use the backward information to enhance its receiver.
 */

#define QOS_FORWARD  0  /* forward (send) traffic information */
#define QOS_BACKWARD 1  /* backward (receive) traffic information */

typedef struct atm_directional_info
{
    UINT32  peak_cell_rate;             /* Cell per Second */
    UINT16  max_sdu_size;               /* Bytes */
    UINT8   cell_loss_ratio;
```

```
   UINT8   filler;
   union
   {
      struct  /* Constant Bit Rate */
      {
         UINT32  filler [2];
         UINT16  cell_transfer_delay;    /* Milliseconds */
      }
      cbr;

      struct  /* Variable Bit Rate (Real Time) */
      {
         UINT32  sustained_cell_rate;    /* Cell per Second */
         UINT32  burst_size;             /* Number of Cells */
         UINT16  cell_transfer_delay;    /* Milliseconds */
         UINT16  cell_delay_variation;   /* Microseconds */
      }
      vbr_rt;

      struct  /* Variable Bit Rate (Non Real Time) */
      {
         UINT32  sustained_cell_rate;    /* Cell per Second */
         UINT32  burst_size;             /* Cells */
         UINT16  cell_transfer_delay;    /* Milliseconds */
      }
      vbr_non_rt;

      struct  /* Available Bit Rate */
      {
         UINT32 minimum_cell_rate;       /* Cell per Second */
      }
      abr;

      struct  /* Unspecified Bit Rate */
      {
         no_additional_info;
      }
      ubr;
   }
   info;
}
ATM_DIRECTIONAL_INFO;

typedef struct atm_qos_rec
{
   UINT8   rate_type;      /* CBR, VBR_RT, VBR, ABR or UBR */
   UINT8   rate_adjustment;
   UINT8   priority;       /* VC priority (0..7) */

   UINT8   filler [1];     /* keep structure aligned */

   ATM_DIRECTIONAL_INFO    /* Directional information */
         direction [2];  /* Forward/backward traffic info */
```

```
}
ATM_QOS_REC;

#endif /* ATMQOS_H */
```

ATMHSMIF.H

```
/*
 * (C) Copyright 1994-1996 Novell, Inc.  All Rights Reserved.
 *
 * File Name       : atmhsmif.h
 * Author          : Henk J. Bots
 * Created On      : Fri, Jan 28 (14:14:12) 1994
 * Last Modified By: Henk J. Bots
 * Last Modified On: Wed, May 22 (17:17:56) 1996
 * Update Count    : 255
 * Status          : ATM TSM provided External Interfaces for HSM's
 */

/* -----------------------Begin Module Header ------------------------------

$Header:   K:/proj/atm/include/vcs/atmhsmif.h_v   1.14   31 May 1996 18:24:16   HBOTS  $

$Project: Novell MPR $

---------------------------------------------------------------------------

   This program is an unpublished copyrighted work which is proprietary to
   Novell, Inc.   and contains confidential information  that is not to be
   reproduced or disclosed  to any  other person  or entity  without prior
   written consent from Novell, Inc.  in each and every instance.

   WARNING:  Unauthorized  reproduction  of  this   program  as  well  as
   unauthorized  preparation of derivative works based upon the program or
   distribution of copies by sale, rental, lease or lending are violations
   of federal  copyright laws and  state trade secret  laws, punishable by
   civil and criminal penalties.

----------------------------------------------------------------------------

$Log:   K:/proj/atm/include/vcs/atmhsmif.h_v  $
 *
 *    Rev 1.14   31 May 1996 18:24:16   HBOTS
 * Remove usage of old "ecb.h", in favor of "odi.h". Add some more
 * debugging for Call Ref and VPCI owner.
 *
 *    Rev 1.13   21 Nov 1995 11:14:02   HBOTS
 * Remove "atmaddr.h" usage for HSMs.
 *
 *    Rev 1.12   10 Nov 1995 13:13:52   HBOTS
 * Add BusTag to HSM registration. Adapt to MSM/NBI usage.
 *
 *    Rev 1.11   02 Nov 1995 13:45:36   HBOTS
 * Update Copyright message to include 1996.
 *
 *    Rev 1.10   15 Sep 1995 13:00:28   HBOTS
 * Correct problem with Watcom 10.5 compiler.
 *
 *    Rev 1.8   14 Sep 1995 18:03:52   HBOTS
 * Prevent Server crash when driver asks for zero length shared RAM
 * for a non-zero address.
 *
 *    Rev 1.7   04 Aug 1995 08:56:46   HBOTS
 * Add HSM Shutdown procedure (called per adapter).
 *
 *    Rev 1.5   25 Jul 1995 11:50:56   HBOTS
 * Add API atmtsm_get_maximum_ecb_size.
 *
 *    Rev 1.4   09 Jun 1995 17:28:06   HBOTS
 * Pass cell_loss_ratio to HSM for PVCs. Add Send OAM Cell to HSM i/f.
 *
 *    Rev 1.3   24 May 1995 18:53:30   HBOTS
 * Use ATMECB (struct _ECB_ or struct ECB, based on ODI.H or ECB.H).
 *
 *    Rev 1.2   23 May 1995 16:13:30   HBOTS
 * Use ODI.H instead of ECB.H.  Make preparations for C-ODI.
 *
 *    Rev 1.1   19 May 1995 15:10:00   HBOTS
 * Add release reason for insufficient VCs.

$EndLog: $

--------------------------- End Module Header --------------------------- */

#ifndef ATMHSMIF_H
#define ATMHSMIF_H

#include "atmqos.h"

#ifndef _ODI_Include_
#include <odi.h>
#endif
#define ATMECB struct _ECB_

/*
 * R E G I S T R A T I O N   V E R S I O N   N U M B E R
 * ====================================================
 *
 * When the HSM registers with the ATM TSM, it must pass the following
 * version  number.   The  TSM  will  reject the registration when the
 * version number it has been built with  does not match and it cannot
 * support the older version.  The oldest version number indicates how
 * much backward compatibility TSM supports.
 */

#define ATMTSM_HSM_VERSION          5
#define ATMTSM_HSM_OLDEST_VERSION    4

/*
 * The  following  are definitions of the two receive entry points the
 * TSM provides to the HSM.
 */

typedef void ATMHSM_TSM_RCV_PROC
                    (ATM_HANDLE tsm_vc_handle, ATMECB* ecb_ptr);
typedef ATMECB* ATMHSM_TSM_RCV_GET_ECB_PROC
                    (ATM_HANDLE tsm_vc_handle, ATMECB* ecb_ptr);

typedef int ATMHSM_ISR_PROC         (ATM_HANDLE hsm_ads_handle);
typedef void ATMHSM_POLL_PROC       (ATM_HANDLE hsm_ads_handle);
typedef void ATMHSM_PERIODIC_PROC   (ATM_HANDLE hsm_ads_handle);

/*
 * Data qualifying information
 * ==========================
 *
 * ATM  defines  a  padding  word of 16 bits (CPCS/U) to fill the last
 * cell of an AAL5 frame.  Some Applications may want to use this word
 * to relay extra information across  the  VC.   The  following  macro
 * allows  the  Application  and  the HSM to pass that information (by
 * using the Driver Workspace bytes 0 and 1).  Applications that don't
 * care to use this do not have to fill out these bytes,  thus sending
 * garbage across.
 */

#ifdef _ODI_Include_
#define CPCS_U( ecb_ptr )                         \
   ((ecb_ptr)->ECB_DriverWorkspace.DWs_i16val [0])
#else
#define CPCS_U( ecb_ptr )                         \
   *((UINT16*) &((ecb_ptr)->driverWorkspace [0]))
#endif

/*
 * An congestion flag is passed with incoming data from the HSM to the
 * User Application.
 */

#define RECEIVE_CONGESTION  0x01

#ifdef _ODI_Include_
#define RECEIVE_FLAGS( ecb_ptr )                  \
   ((ecb_ptr)->ECB_DriverWorkspace.DWs_i8val [2])
#else
#define RECEIVE_FLAGS( ecb_ptr )    \
   (ecb_ptr)->driverWorkspace [2])
```

```
#endif

/*
 * Interrupt and Multiprocessor Lock definitions
 * ==============================================
 */

/*
 * The  user  must  allocate  the  space for each lock,  and guarantee
 * proper alignment (on a memory word  boundary).   In  case of NW386,
 * 32-bit alignment is required.
 *
 * There  are  two sets of lock calls,  one where the TSM will disable
 * interrupts,  as well as acquiring the lock,  another where just the
 * lock is obtained.   The  latter have a suffix of "_ni",  and assume
 * that the HSM has interrupts disabled.
 */

typedef struct atmtsm_lock
{
   UINT32  lock;
   UINT32  interrupt_flags;
}
ATMTSM_LOCK;

/*
 *   T H E   H S M _ V C _ H A N D L E ,   A N D   O T H E R   I N F O
 *   ==================================================================
 *
 * The HSM can return the hsm_vc_handle in one of the following ways:
 *
 * -   When   returning  from  control  ATM_TSM_TO_HSM_ENABLE_VC,   by
 * inserting  the   hsm_vc_handle  in  field  "hsm_vc_handle"  of  the
 * "atmhsm_control".enable_vc.
 *
 * - Upon passing the control ATM_HSM_TO_TSM_VC_ENABLED to the TSM. If
 * the HSM operates in this way, it may be possible that the TSM calls
 * the   HSM  with  the  control  ATM_TSM_TO_HSM_DISABLE_VC,   with  a
 * NULL_ATM_HANDLE.  The HSM must be able to accept this control,  and
 * use the VPI_VCI number to identify the virtual connection that must
 * be disabled (before it has been completely enabled).
 *
 * The  TSM  will  not  pass output to the HSM,  until the VC is fully
 * enabled (the 2-way handshake  is  completed).  Conversely,  the HSM
 * must not pass input until the VC is fully enabled either.
 *
 * The  TSM may call the HSM during processing the positive reply to a
 * Contract Verification (the ATM_HSM_TO_TSM_CONTRACT_VERIFIED),  with
 * a ATM_TSM_TO_HSM_ENABLE_VC.
 *
 * Both  the  ATM_HSM_TO_TSM_DISABLE_VC  and ATM_TSM_TO_HSM_DISABLE_VC
 * do not expect a reply (are unilateral).
 */
```

```
/*
 *   A T M _ H S M _ T O _ T S M _ C O N T R O L
 *   ===========================================
 *
 * The  following  defines the Control information passed from the HSM
 * to the TSM.  The HSM,  upon registration,  passes the Control Entry
 * point (procedure address to be called by the TSM) to the TSM.
 *
 * Note  that  HSM cannot allocate memory until after the registration
 * is complete.   It  also  may need the registered hardware to obtain
 * such things as the End Station Identifier (ESI).  Therefore,  after
 * registration but before HSM  starts  the  interface,  it can update
 * such control information (ATM_HSM_TO_TSM_UPDATE_CONTROL).
 */

#define ATM_HSM_TO_TSM_REGISTER_HW       1
#define ATM_HSM_TO_TSM_DEREGISTER_HW     2
#define ATM_HSM_TO_TSM_DEREGISTER        3
#define ATM_HSM_TO_TSM_START_INTERFACE   4
#define ATM_HSM_TO_TSM_STOP_INTERFACE    5
#define ATM_HSM_TO_TSM_CONTRACT_VERIFIED 6
#define ATM_HSM_TO_TSM_VC_ENABLED        7
#define ATM_HSM_TO_TSM_DISABLE_VC        8
#define ATM_HSM_TO_TSM_STOP_OUTPUT       9   /* Begin Output Flow Control */
#define ATM_HSM_TO_TSM_START_OUTPUT     10   /* End Output Flow Control */

/*
 * The following defines the information passed during registration of
 * the hardware. For future extention, make sure all unused fields are
 * zero.
 */

typedef struct
{
    char*   transmission_type;      /* e.g. SDH, SONETSTSx, IBM */
    char*   media_type;             /* e.g. FIBER, COAX, UTP5 */
    UINT32  speed;                  /* speed in Kbps/Mbps/Gbps */
    UINT8   speed_base;             /* 0=Kbps, 1=Mbps, 2=Gbps */
    UINT8   upper_vpc;              /* highest Virtual Path Conn nr */
    UINT16  upper_vcc;              /*   "  Virtual Channel Conn nr */
    UINT32  max_cell_rate;          /* Cells per Second */
    UINT8   max_physical_fragments; /* atmtsm_logical_to_physical */
    UINT8   reserved_1 [3];
    void*   bus_tag;                /* Null or from CMSMGetBusTag */
    UINT8   esi [6];
    UINT8   reserved_2 [14];
    ATMHSM_ISR_PROC*
        hsm_isr_proc;               /* interrupt service routine */
    UINT16  sharing_flags;
    UINT16  slot_number;
    UINT16  channel_number;
    UINT16  port0;
```

```
   UINT16  port0_range;
   UINT16  port1;
   UINT16  port1_range;
   UINT8   interrupt_number;       /* (IRQ) 0xff means none used */
   UINT8   dma_line;              /* 0xff means none used */
   void*   memory0_address;        /* shared RAM physical address */
   UINT32  memory0_bytes;          /* shared RAM size */
   void*   memory1_address;
   UINT32  memory1_bytes;
   void*   linear0_address;        /* Ret: shared RAM logical addr */
   void*   linear1_address;
}
ATM_HSM_HW_INFO;

typedef struct atm_hsm_to_tsm_control
{
   UINT32      control_type;
   ATM_HANDLE  tsm_ads_handle;
   union
   {
      struct         /* REGISTER_HW */
      {
         ATM_HSM_HW_INFO*    hw_info_ptr;

      } register_hw;

      struct         /* DEREGISTER_HW */
      {
         no_additional_info;

      } deregister_hw;

      struct         /* DEREGISTER */
      {
         no_additional_info;

      } deregister;

      struct         /* START_INTERFACE */
      {
         no_additional_info;

      } start_interface;

      struct         /* STOP_INTERFACE */
      {
         no_additional_info;

      } stop_interface;

      struct         /* CONTRACT_VERIFIED */
      {
         ATM_HANDLE  tsm_contract_handle;/* set by HSM, as given by TSM */
```

```
            ATM_QOS_REC qos;                /* quality of service */
            UINT8       return_code;        /* 0 when accepted by HSM */

        } contract_verified;

        struct          /*  V C _ E N A B L E D  */
        {
            ATM_HANDLE  tsm_vc_handle;  /* set by HSM, as given by TSM */
            ATM_HANDLE  hsm_vc_handle;  /* set by HSM */

        } vc_enabled;

        struct          /*  D I S A B L E _ V C  */
        {
            ATM_HANDLE  tsm_vc_handle;
            UINT8       reason_code;

        } disable_vc;

        struct          /*  S T O P _ O U T P U T  */
        {
            ATM_HANDLE  tsm_vc_handle;

        } stop_output;

        struct          /*  S T A R T _ O U T P U T  */
        {
            ATM_HANDLE  tsm_vc_handle;

        } start_output;

    } info;
}
ATM_HSM_TO_TSM_CONTROL;

/*
 * Errors returned from atmtsm_hsm_control
 * ---------------------------------------
 */

#define ATM_HSM_CTRL_COMPLETE                       0
#define ATM_HSM_CTRL_BAD_TSM_ADS_HANDLE             -1
#define ATM_HSM_CTRL_INVALID                        -2
#define ATM_HSM_CTRL_HARDWARE_REGISTER_FAILURE    -3
#define ATM_HSM_CTRL_BAD_START_STOP_INTERFACE     -4
#define ATM_HSM_CTRL_INTERFACE_MUST_BE_STOPPED    -6
#define ATM_HSM_CTRL_HW_ALREADY_REGISTERED        -6 /* as previous */
#define ATM_HSM_CTRL_BAD_TSM_VC_HANDLE              -7

/*
 * Register Hardware sharing flags
 * -------------------------------
 */
```

```
#define ATM_SHARE_PORT0     0x0002
#define ATM_SHARE_PORT1     0x0004
#define ATM_SHARE_MEMORY0   0x0008
#define ATM_SHARE_MEMORY1   0x0010
#define ATM_SHARE_INTERRUPT 0x0020
#define ATM_SHARE_DMA       0x0080

/*
 *   A T M _ T S M _ T O _ H S M _ C O N T R O L
 *   ==========================================
 *
 * The  following  defines the Control information passed from the TSM
 * to the HSM.  The HSM,  upon registration,  passes the Control Entry
 * point (procedure address to be called by the TSM) to the TSM.
 */

#define ATM_TSM_TO_HSM_VERIFY_CONTRACT  1
#define ATM_TSM_TO_HSM_ENABLE_VC        2
#define ATM_TSM_TO_HSM_DISABLE_VC       3
#define ATM_TSM_TO_HSM_CONTROL_CHANGE   4
#define ATM_TSM_TO_HSM_SEND_OAM_CELL    5

typedef struct atm_tsm_to_hsm_control
{
   UINT32      control_type;
   ATM_HANDLE  hsm_ads_handle;
   union
   {
      struct         /*  V E R I F Y _ C O N T R A C T  */
      {
         ATM_HANDLE  tsm_contract_handle;
         ATM_QOS_REC qos;           /* quality of service */

      } verify_contract;

      struct         /*  E N A B L E _ V C  */
      {
         ATM_HANDLE  tsm_vc_handle;
         UINT32      vpi_vci;       /* set by TSM */
         ATM_QOS_REC qos;           /* quality of service */
         ATM_HANDLE  hsm_vc_handle;  /* set to NULL_ATM_HANDLE (TSM),
                                 optionally returned by HSM */
         ATMHSM_TSM_RCV_PROC*        tsm_rcv_proc;
         ATMHSM_TSM_RCV_GET_ECB_PROC* tsm_rcv_get_ecb_proc;

      } enable_vc;

      struct         /*  D I S A B L E _ V C  */
      {
         UINT32      vpi_vci;
         ATM_HANDLE  hsm_vc_handle;  /* passed by TSM (maybe */
                               /*   NULL_ATM_HANDLE)   */
```

```
        } disable_vc;

        struct          /*  C O N T R O L _ C H A N G E  */
        {
           ATM_HANDLE                  hsm_vc_handle;
           ATMHSM_TSM_RCV_PROC*         tsm_rcv_proc;
           ATMHSM_TSM_RCV_GET_ECB_PROC* tsm_rcv_get_ecb_proc;

        } control_change;

        struct          /*  S E N D _ O A M _ C E L L  */
        {
           ATM_HANDLE  hsm_vc_handle;  /* VC to send OAM Cell on */
           ATMECB*     oam_cell_ecb_ptr;

        } send_oam_cell;

    } info;
}
ATM_TSM_TO_HSM_CONTROL;

/*
 * Contract Verify return codes
 * ----------------------------
 */

#define HSM_CONTRACT_ACCEPTED              0
#define HSM_CONTRACT_CANNOT_DO_RATE        1

/*
 * Disable VC reason codes
 * -----------------------
 */

#define HSM_DISABLE_UNDEFINED       0
#define HSM_DISABLE_TOO_MANY_VCS    1

/*
 * Receive Data Error Types (atmtsm_rcv_error/atmtsm_rcv_vc_error)
 * ---------------------------------------------------------------
 */

#define RCV_ERROR_DISABLED_VCI  1   /* through "atmtsm_rcv_error" only */
#define RCV_ERROR_CRC           2
#define RCV_ERROR_TIMEOUT       3   /* Missing cells */
#define RCV_ERROR_PDU_TOO_LARGE 4

/*
 * Macro to get the Physical Receive Address for the Receive ECB
 * -------------------------------------------------------------
 *
 * The  physical  address  is  inserted  by the ATMTSM when the ECB is
 * obtained,  and thus matches the logical  address  as stored at that
 * time.
 */

#ifdef _ODI_Include_
#define PHYSICAL_RCV_ADDRESS( ecb_ptr ) \
       (*(UINT8**) &(ecb_ptr)->ECB_ProtocolWorkspace.PWs_pval [1])
#else
#define PHYSICAL_RCV_ADDRESS( ecb_ptr ) \
       (*(UINT8**) &(ecb_ptr)->socketNumber)
#endif

/*
 * Logical to Physical Fragment list definitions
 * ---------------------------------------------
 *
 * The following are definitions used by the HSM and TSM to define the
 * memory fragment lists passed to "atmtsm_logical_to_physical".
 */

typedef struct atmtsm_fragment
{
   void*   address;
   INT32   size;
}
ATMTSM_FRAGMENT;

typedef ATMTSM_FRAGMENT ATMTSM_FRAGMENT_LIST [];

/*
 *   A T M T S M _ R E G I S T E R _ H S M
 *   ====================================
 *
 * The  following  defines the structure to be used by the HSM when it
 * registers with the ATM TSM.
 */

typedef void ATMTSM_HSM_CONTROL_PROC  (struct atm_tsm_to_hsm_control*
                              h2t_cptr);
typedef void ATMTSM_HSM_SEND_PROC     (ATM_HANDLE hsm_vc_handle,
                              ATMECB* ecb_ptr);
typedef void ATMTSM_HSM_SHUTDOWN_PROC (ATM_HANDLE hsm_ads_handle);

typedef UINT8* ATM_CUSTOM_TEXT_LINE;
typedef ATM_CUSTOM_TEXT_LINE ATM_CUSTOM_STATS_LIST [];

/*
 * The  following  values  are returned by the TSM after registration.
 * The value is platform dependend.   A  NetWare  Server  returns FAIR
 * since  it may run certain applications that sometimes do not  yield
 * for 10 Milliseconds or more.  A NetWare Client (under NIOS) returns
 * LIMITED because  it  depends on timer interrupts.  UnixWare returns
 * FAIR because it uses 1 Millisecond timer interrupts. Good indicates
 * that  the  OS  polls are truely  based  on  inversed  load  of  the
 * processor(s).
 */

#define POLL_SUPPORT_LEVEL_NONE     0
#define POLL_SUPPORT_LEVEL_LIMITED  1   /* NetWare Client */
#define POLL_SUPPORT_LEVEL_FAIR     2   /* NetWare Server */
#define POLL_SUPPORT_LEVEL_GOOD     3

typedef struct atmtsm_register_hsm_control
{
   /* Values passed by the HSM to the TSM */

   UINT32      version;           /* must be ATMTSM_HSM_VERSION */
   struct LoadDefinitionStructure**
            hsm_nlm_handle_ptr; /* passed by NetWare to HSM, */
                           /* must point to stack variable */
   ATMTSM_HSM_CONTROL_PROC*
            hsm_control_proc;
   ATMTSM_HSM_SEND_PROC*
            hsm_send_proc;      /* driver send routine */
   char*       card_name;         /* (long) name of card */
   char*       driver_name;       /* (short) name of driver */
   char*       board_name;        /* up to 17 characters */
   UINT16      driver_options;
   UINT16      max_frame_size;     /* 512..65535 */
   ATM_CUSTOM_STATS_LIST*          /* NULL indicates no such Stats */
            custom_stats_ptr;
   UINT32      hsm_ads_size;       /* size of HSM Adapter Data Space */
   ATM_HANDLE  hsm_ads_handle;     /* assigned by TSM or HSM */
   ATMTSM_HSM_SHUTDOWN_PROC*
            hsm_shutdown_proc;  /* allows for hardware shutdown */
   UINT8       future_extension_1 [8];     /* preset to zeros */

   /* Values returned by the TSM to the HSM */

   ATM_HANDLE  tsm_ads_handle;
   UINT32      board_number;
   UINT8       poll_support_level; /* 0 .. 3 */
   UINT8       future_extension_2 [15];    /* preset to zeros */
}
ATMTSM_REGISTER_HSM_CONTROL;

/*
 * Various driver options (all undefined option bits must be zero)
 * ---------------------------------------------------------------
 */

/* None are currently defined */

/*
 * Custom Statistics definitions
 * -----------------------------
 *
 * If the HSM uses custom statistics, the following record is an array
 * of pointers to the custom text. There must be one array-element per
 * custom counter, followed by a NULL pointer.  Each text line must be
 * zero-byte terminated,  and not contain more than 41 characters.
 *
 * To increment a custom statistics counter by one,  the HSM must call
 * "atmtsm_incr_custom_counter",   passing the custom statistic number
 * (the first one is zero) and the address  of  the  TSM Adapater Data
 * Space.
 */

/*
 * Example of custom statistics usage:
 *
 *  static ATM_CUSTOM_STATS_LIST custom_stats =
 *  {
 *     "Custom Statistic #0 Errors:",
 *     "Some Other Custom Statistic Errors:",
 *     NULL,
 *  };
 */

/*
 * Various other definitions
 * -------------------------
 */

#define MAX_CARD_NAME          62
#define MAX_DRIVER_NAME         8
#define MAX_TRANSMISSION_TYPE  23
#define MAX_MEDIA_TYPE         23

#define SPEED_BASE_KBPS        0
#define SPEED_BASE_MBPS        1
#define SPEED_BASE_GBPS        2

/*
 * Errors returned from atmtsm_register_hsm
 * ----------------------------------------
 */

#define ATM_HSM_REG_COMPLETE                    0
#define ATM_HSM_REG_VERSION_MISMATCH           -1
#define ATM_HSM_REG_NO_RESOURCES               -2
#define ATM_HSM_REG_LSL_REGISTER_FAILURE       -3
#define ATM_HSM_REG_FIRMWARE_LOAD_FAILED       -4

/*
 *  A T M T S M   E N T R Y - P O I N T S   F O R   H S M
 *  =====================================================
 *
 * After  successful VC Enabling,  there will be a tsm_vc_handle and a
 * hsm_vc_handle.  The HSM defines the value of the hsm_vc_handle.  It
 * may be the  vpi_vci  number,   or  the  address of a HSM defined VC
 * Control Block,  or anything else.
 *
 * When the HSM requests an ECB (through atmtsm_get_ecb),  the ATM TSM
 * will preset field ecb_ptr->fragmentDescriptor [0].address.  The HSM
 * must store the first byte of the incoming frame at this address (in
 * contiguous memory).
 *
 * When the HSM passes input received for a specific VC,  it must pass
 * the correct tsm_vc_handle. Receive ECBs are not associated with any
 * VC  at the time the HSM requests one.  The receive entry points are
 * passed  to  the  HSM  during  the ESTABLISH VC command,  and may be
 * changed by the TSM at a later time.
 *
 * Upon deregistration, the HSM must pass the tsm_ads_handle (same one
 * that was assigned during registration)
 */

extern long atmtsm_register_hsm (ATMTSM_REGISTER_HSM_CONTROL*);
extern long atmtsm_hsm_control  (ATM_HSM_TO_TSM_CONTROL*);
extern void atmtsm_poll_address (ATM_HANDLE tsm_ads_handle,
            ATMHSM_POLL_PROC* hsm_poll_proc,
            UINT32 minimum_microseconds_delay);
extern void atmtsm_periodic_call_address (ATM_HANDLE tsm_ads_handle,
            ATMHSM_PERIODIC_PROC* hsm_periodic_proc,
            UINT32 milliseconds);
extern UINT8* atmtsm_get_firmware (ATM_HANDLE tsm_ads_ptr,
            UINT32* firmware_length_ptr);
extern UINT32 atmtsm_micro_delay (long microseconds);
extern UINT32 atmtsm_get_maximum_ecb_size (void);
extern ATMECB* atmtsm_get_ecb (ATM_HANDLE tsm_ads_handle);
extern void atmtsm_return_ecb (ATMECB* ecb_ptr);
extern void atmtsm_send_complete (ATMECB* ecb_ptr);
extern void atmtsm_rcv_oam_cell (ATM_HANDLE tsm_vc_handle, ATMECB* ecb_ptr);
extern void atmtsm_rcv_error (ATM_HANDLE tsm_ads_handle, ATMECB* ecb_ptr,
            UINT8 error_type);
extern void atmtsm_rcv_vc_error (ATM_HANDLE tsm_vc_handle, ATMECB* ecb_ptr,
            UINT8 error_type);
extern void atmtsm_incr_custom_counter (ATM_HANDLE tsm_ads_handle,
            UINT8 custom_counter_number);
extern void atmtsm_update_custom_counter (ATM_HANDLE tsm_ads_handle,
            UINT8 custom_counter_number, long update_value);
extern void atmtsm_set_custom_counter (ATM_HANDLE tsm_ads_ptr,
            UINT8 custom_counter_number, UINT32 value);
extern int  atmtsm_printf (const char* format, ...);
extern void* atmtsm_alloc (ATM_HANDLE tsm_ads_handle, UINT32 bytes);
extern void  atmtsm_free (void* mem_ptr);
extern UINT32 atmtsm_logical_to_physical_list (ATM_HANDLE tsm_ads_handle,
            INT8 logical_fragments, ATMTSM_FRAGMENT* logical_list,
            ATMTSM_FRAGMENT* physical_list);
extern void* atmtsm_logical_to_physical (void* logical_address);
extern void* atmtsm_physical_to_logical (void* physical_address);
extern UINT8 atmtsm_register_lock (struct atmtsm_lock* lock_ptr,
            char* lock_id);
extern void atmtsm_deregister_lock (struct atmtsm_lock* lock_ptr);
extern void atmtsm_lock (struct atmtsm_lock* lock_ptr);
extern void atmtsm_lock_ni (struct atmtsm_lock* lock_ptr);
extern UINT8 atmtsm_conditional_lock (struct atmtsm_lock* lock_ptr);
extern UINT8 atmtsm_conditional_lock_ni (struct atmtsm_lock* lock_ptr);
extern void atmtsm_unlock (struct atmtsm_lock* lock_ptr);
extern void atmtsm_unlock_ni (struct atmtsm_lock* lock_ptr);

#endif /* ATMHSMIF_H */
```

ATMHSM.IMP

```
#
# (C) Copyright 1995-1996 Novell, Inc.  All Rights Reserved.
#
# File Name       : atmhsm.imp
# Author          : Henk J. Bots
# Created On      : Tue, Apr  4 (10:04:12) 1995
# Last Modified By: Henk J. Bots
# Last Modified On: Tue, May 21 (18:33:24) 1996
# Update Count    : 12
# Status          : ATMTSM entry points for HSMs
#
#
# $Header:   K:/proj/atm/imports/vcs/atmhsm.imv   1.7   21 May 1996 18:33:38   HBOTS  $
#
# $Project: Novell MPR ATM Development $
#
# $Creator: Henk J. Bots $
#
# ---------------------------------------------------------------------------
#
# This program is an unpublished copyrighted work which is proprietary
# to Novell, Inc. and contains confidential information that is not
# to be reproduced or disclosed to any other person or entity without
# prior written consent from Novell, Inc. in each and every instance.
#
# WARNING:  Unauthorized reproduction of this program as well as
# unauthorized preparation of derivative works based upon the
# program or distribution of copies by sale, rental, lease or
# lending are violations of federal copyright laws and stat trade
# secret laws, punishable by civil and criminal penalties.
#
# ---------------------------------------------------------------------------
#
# $Log:   K:/proj/atm/imports/vcs/atmhsm.imv  $
#
#    Rev 1.7   21 May 1996 18:33:38   HBOTS
#  Remove last comma.
#
#     Rev 1.6   20 Feb 1996 10:09:34   HBOTS
#  Add CMSM import symbols available to ATM HSMs.
#
#     Rev 1.3   02 Nov 1995 13:45:44   HBOTS
#  Update Copyright message to include 1996.
#
        atmtsm_register_hsm,
        atmtsm_hsm_control,
        atmtsm_poll_address,
        atmtsm_periodic_call_address,
        atmtsm_micro_delay,
        atmtsm_get_firmware,
        atmtsm_get_maximum_ecb_size,
        atmtsm_get_ecb,
        atmtsm_return_ecb,
        atmtsm_send_complete,
        atmtsm_printf,
        atmtsm_alloc,
        atmtsm_free,
        atmtsm_logical_to_physical_list,
        atmtsm_logical_to_physical,
        atmtsm_physical_to_logical,
        atmtsm_rcv_oam_cell,
        atmtsm_rcv_error,
        atmtsm_rcv_vc_error,
        atmtsm_incr_custom_counter,
        atmtsm_update_custom_counter,
        atmtsm_set_custom_counter,
        atmtsm_register_lock,
        atmtsm_deregister_lock,
        atmtsm_lock,
        atmtsm_conditional_lock,
        atmtsm_unlock,
        atmtsm_lock_ni,
        atmtsm_conditional_lock_ni,
        atmtsm_unlock_ni,
#
# CMSM provided entry points (include "cmsm.h").  These entry points
# are safe to be used by non LAN-ODI compliant HSMs.
#
        CMSMGetAlignment,
        CMSMGetBusInfo,
        CMSMGetBusName,
        CMSMGetBusTag,
        CMSMGetBusType,
        CMSMGetBusSpecificInfo,
        CMSMGetCardConfigInfo,
        CMSMGetCurrentTime,
        CMSMGetInstanceNumber,
        CMSMGetInstanceNumberMapping,
        CMSMGetNBIConfiguration,
    CMSMGetSlot,
    CMSMGetSlotName,
    CMSMGetUniqueIdentifier,
    CMSMGetUniqueIdentifierParameters,
    CMSMGetMicroTimer,
    CMSMSearchAdapter,
    CMSMScanBusInfo,
    CMSMReadPhysicalMemory,
    CMSMWritePhysicalMemory,
    CMSMRdConfigSpace8,
    CMSMRdConfigSpace16,
    CMSMRdConfigSpace32,
    CMSMWrtConfigSpace8,
    CMSMWrtConfigSpace16,
    CMSMWrtConfigSpace32,
    CMSMYieldWithDelay
```

PARSER.H

```
/*
 * (C) Copyright 1995 Novell, Inc., All Rights Reserved.
 *
 * File Name       : parser.h
 * Author          : Henk J. Bots
 * Created On      : Mon, Jul 15 (14:26:22) 1993
 * Last Modified By: Henk J. Bots
 * Last Modified On: Tue, Jun  4 (09:46:18) 1996
 * Update Count    : 85
 * Status          : Unknown
 */

/* -----------------------Begin Module Header ------------------------------

$Header:   K:/proj/parser/include/vcs/parser.h_v   1.6   04 Jun 1996 09:46:32   HBOTS  $

$Project: Novell Multiprotocol Brouter $

$Locker: $

---------------------------------------------------------------------------

   This program is an unpublished copyrighted work which is proprietary to
   Novell, Inc.   and contains confidential information  that is not to be
   reproduced or disclosed  to any  other person  or entity  without prior
   written consent from Novell, Inc.  in each and every instance.

   WARNING:  Unauthorized  reproduction  of  this   program  as  well  as
   unauthorized  preparation of derivative works based upon the program or
   distribution of copies by sale, rental, lease or lending are violations
   of federal  copyright laws and  state trade secret  laws, punishable by
   civil and criminal penalties.
```

```
----------------------------------------------------------------------------

$Log:   K:/proj/parser/include/vcs/parser.h_v  $
 *
 *    Rev 1.6   04 Jun 1996 09:46:32   HBOTS
 * Add more information to the sample parse code.
 *
 *    Rev 1.5   25 Sep 1995 11:36:18   HBOTS
 * Make usage of CLIB optional (by using ImportPublicSymbol).
 *
 *    Rev 1.3   10 May 1995 18:07:10   HBOTS
 * Add some more useful comments, and GET_COMMAND_TEXT_PTR macro.
 *
 *    Rev 1.2   25 Apr 1995 12:02:36   HBOTS
 * Minor modification for ANSI-C compatibility.
 *
 *    Rev 1.1   17 Apr 1995 12:53:36   HBOTS
 * Improve sample code comments.
 *
 *    Rev 1.0   13 Apr 1995 17:23:02   HBOTS
 * Initial version of PARSER at its own project directory.
 *
$EndLog: $

-------------------------- End Module Header --------------------------- */

#ifndef PARSER_H
#define PARSER_H

/*
 * When  calling  PARSER_INITIALIZE,   the  PARSER_IF_VERSION  must be
 * passed.   This allows the parser to verify compatibility.  An error
 * will be returned when the versions are incompatible. If the version
 * is between the current and the minimum,  the parser will accomodate
 * the differences.
 */

#define PARSER_IF_VERSION         5 /* More exported symbols */
#define PARSER_MINIMUM_IF_VERSION 3

#if 0
/**********************************************************************/
/********************* SAMPLE COMMAND PARSER ***************************/
/**********************************************************************/

/*
 * The  following  is  a  sample a command parsing program,  using the
 * parser defined tables and  interfaces.  Note that most of the table
 * macros initialize internally used table parts, thus the omission of
 * semi-colons is needed.
 *
 * The parser is not case sensitive. All entered input is converted to
 * upper-case,  with  the  of quote embedded Names.
 *
 * Parameter  must  be  separated by a comma and/or one or more spaces
 * (or tabs).  Multiple spaces/tabs/commas  are  treated  as  a single
 * space.
 *
 * The parser can check integer ranges, minimum and maximum lengths of
 * names,  upper  byte  value  of  a given address (e.g.  for a 46 bit
 * address).
 *
 * A  boolean  parameter  may  be  defined  by just its parameter name
 * (indicating a TRUE).
 *
 * A list of values must either be enclosed in parentheses "(1 2 3)" or
 * brackets "{ 1 2 4 }".
 *
 *
 * Example of some possible parameters and values:
 * ----------------------------------------------
 *
 * Integer value: KW = 2, KW=0xff, KW=-7, KW=(0x22, 00012 56)
 *
 * Hex value: KW=0ff00, KW=100  KW=f19b
 *         (parser will optionally perform range checking)
 *
 * Boolean value: KW, KW=true, KW=off, KW=yes, KW = on
 *
 * Name: KW=target_name, KW = "Name of thing", KW={n1,n2}
 *         (parser will perform min/max length checking)
 *
 * Equate: KW=ieee, KW=dec, KW=none
 *
 * Address (hex): KW=1b-2-0-5a, KW=00001b02005a, KW=$d8-40-0-5a
 *         ($ prefix indicates canonical conversion)
 *
 * Address (decimal): KW=128.55.12.14  KW=0x80.55.12.14 KW=1700.62000
 *         (parser will check upper value of individual elements)
 *
 * Multi-Address: KW=(36:a00:1200:5:b2 1b-22-4-a6 1)
 *
 * Dual types: KW=55, KW=off (user specifies which two types are
 *           allowed, hopefully preventing ambiguities, e.g. not
 *           mixing names and alpha equates)
 *
 * The user has the option to Preview selected parameters and possibly
 * modify  the  associated values.  This may be useful for things like
 * NODE=1b2578L,   where the L needs to be removed prior to converting
 * the address.  The  user can convert the value to $1b2578 (which the
 * parser will use upon return to do canonical conversion).
 */

UINT32 parse_my_command (UINT8* command_text)

{
   static initial_load = TRUE;

   /*
    * The  following  are  the  user  defined values that the parser will
    * return from ParserGetNextParameterAndValue (as user_id).  It can be
    * used to identify the given parameter.
    */

#define PARM_STP    1
#define PARM_T      2
#define PARM_FS     3
#define PARM_VRN    4
#define PARM_NODE   5
#define PARM_IPA    6
#define PARM_LL     7
#define PARM_AA     8
#define PARM_DEBUG  9

   /*
    * NOTE  that  all  table  entries  must  be  in  upper case (with the
    * exception  of  user help and dimensions),  otherwise  the  user  is
    * forced to enter the  Equated  value  as  a  string (KW="none") (the
    * parser will internally convert the (non-string)  user  entered data
    * to upper case prior to the various table lookups).
    */

   static PV_EQUATE pvt_stp =
   {
      { "IEEE", 1 },
      { "DEC", 2 },
      { "NONE", 0 },
      { "OFF", 0 },
      { NULL }, /* end of table */
   };
   static PV_COMPLEX_ADDRESS pvt_aa [] =
   {
      PVE_COMPLEX_ADDRESS (HEX_ADDRESS, 13 * 8, ':', WORD_SEPARATOR)
      PVE_COMPLEX_ADDRESS (HEX_ADDRESS,  6 * 8, '-', BYTE_SEPARATOR)
      PVE_COMPLEX_ADDRESS (HEX_ADDRESS,  1 * 8, '\0', NO_SEPARATOR)
      PVE_COMPLEX_ADDRESS_END
   };
   static UINT8* d_bytes = "Bytes";    /* For Help purposes only */
   static PV_INT_RANGE pvt_fs = { 512, 17800, &d_bytes };
   static PV_INT_RANGE pvt_vrn = { 1, 4095, NULL };
   static PV_EQUATE pvt_vrn_equ =
   {
      { "NONE", 0 },
      { "OFF", 0 },
      { NULL },
   };
   /*
    * NOTE that the preferred order for DUAL TYPES is:
    *   1) PDE_BOOLEAN
```

```
 *   2) PDE_EQUATE
 *   3) PDE_NAME
 *   4) PDE_HEX/INT(_RANGE) / PDE_ADDRESS (don't mix these!)
 */

static PV_DUAL_TYPE dpvt_vrn =
{
   PDE_EQUATE (PFP_NONE, &pvt_vrn_equ)
   PDE_INT_RANGE (PFP_NONE, &pvt_vrn)
};
static UINT8 *fs_help = "\t\t\tDefines maximum negotiated Frame Size\n";
static UINT8 *t_help = "\t\t\tTarget Name is for the Call Manager\n";

PARSER_PARAMETER_TABLE (test_parm_tbl)
   PTE_EQUATE ("SPANNING_TREE_PROTOCOL", "STP", PARM_STP,
      NO_USER_HELP, PFP_NONE, &pvt_stp)
   PTE_NAME ("TARGET", "T", PARM_T,
      &t_help, PFP_NONE, 1, 47)
   PTE_INT_RANGE ("FRAME_SIZES", "FS", PARM_FS,
      &fs_help, PFP_LIST_ALLOWED | PFP_DUPL_ALLOWED, &pvt_fs)
   PTE_DUAL_TYPE ("VIRTUAL_RING_NUMBER", "VRN", PARM_VRN,
      NO_USER_HELP, &dpvt_vrn)
   PTE_ADDRESS ("NODE", NULL, PARM_NODE,
      NO_USER_HELP, PFP_NONE,
      HEX_ADDRESS, 48, '-', BYTE_SEPARATOR)
   PTE_ADDRESS ("IP_ADDRESS", "IPA", PARM_IPA,
      NO_USER_HELP, PFP_REQUIRED,
      DECIMAL_ADDRESS, 32, '.', BYTE_SEPARATOR)
   PTE_INT ("LEARNING_LEVEL", "LL", PARM_LL,
      NO_USER_HELP, PFP_NONE)
   PTE_COMPLEX_ADDRESS ("ATM_ADDRESS", "AA", PARM_AA,
      NO_USER_HELP, PFP_NONE, &pvt_aa)
   PTE_BOOLEAN ("DEBUG", NULL, PARM_DEBUG,
      NO_USER_HELP, PFP_SUPPRESS_HELP)
PARSER_PARAMETER_TABLE_END;

PARSER_COMMAND_TABLE (test_cmd, "LOAD MYCOMMAND", PFC_NONE,
   test_parm_tbl, NULL);

PARSER_INTERFACE pi;

if (!ParserInitialize (&pi, &test_cmd, command_text,
   PARSER_IF_VERSION, NULL))
{
   if (pi.user_id == PUI_HELP_REQUEST)
   {
      if (!initial_load)
         return (0);    /* Help request complete */
   }
   return (1);          /* Error or initial load Help request */
}
while (ParserGetNextParameterAndValue (&pi))
{
        switch (pi.user_id)
        {
        case PARM_STP:
            /* pi_ptr->value.equ contains user defined equated value */
            break;
        case PARM_T:
            /* pi_ptr->value.name contains C-like name string */
            break;
        case PARM_FS:
            /*
             * Note that elements from a list of values can either be
             * obtaines with the following "while" loop,   or by just doing
             * the next ParserGetNextParameterAndValue call, which will come
             * here with the next value for the same parameter.
             */
            do
            {
                /* pi_ptr->value.uint contains frame size */
            }
            while (pi.in_list && ParserGetNextValue (&pi));
            break;
        case PARM_VRN:
            /* pi_ptr->value.uint contains ring number */
            break;
        case PARM_NODE:
        case PARM_IPA:
            /* pi_ptr->value.address contains address bytes, size
               depends on maximum number of bits of address. If user
               entered less, parser will zero fill the upper byte(s). */
            break;
        case PARM_LL:
            /* pi.value.uint/sint contains LL value */
            break;
        case PARM_AA:
            {
                int i;
                for (i = 0; i < PARSER_MAX_ADDRESS; i++)
                    OutputToScreen (screen_ptr, " %02x",
                        pi.value.address [i]);
                OutputToScreen (screen_ptr, "\n");
            }
            break;
        case PARM_DEBUG:
            if (pi.value.boolean)
                EnterDebugger ();
        default:
            return (2); /* error */
            break;
        }
    }
    initial_load = FALSE;
    return (0); /* no errors */
}
```

```
/*
 * The  NetWare  loader  will  call  the  NLM start procedure with (at
 * least) the following parameters:
 */

long nlm_start_proc (struct LoadDefinitionStructure* nlm_handle,
                  void* screen_ptr,
                  UINT8* command_text_ptr)
{

}

/*
 * If macro GET_COMMAND_TEXT_PTR is used,  the NLM start procedure may
 * look as follows:
 */

long nlm_start_proc (struct LoadDefinitionStructure* nlm_handle)
{
   UINT8 command_text_ptr = GET_COMMAND_TEXT_PTR (&nlm_handle);

}

/*
 * When  the  user entered "LOAD xxx.NLM p1=x p2=y",  the command text
 * pointer will  directly point to "p1=x p2=y",  thus that pointer can
 * be passed unchanged to ParserInitialize.
 */

/*************************************************************************/
/***************** END OF SAMPLE COMMAND PARSER ************************/
/*************************************************************************/
#endif

#define LITTLE_ENDIAN    /* do byte swapping things */

#ifndef UINT8
#define UINT8 unsigned char
#endif

#ifndef UINT16
#define UINT16 unsigned short
#endif

#ifndef UINT32
#define UINT32 unsigned long
#endif

#ifndef INT32
#define INT32 long
#endif
```

```
#ifndef NULL
#define NULL 0
#endif

#ifndef FALSE
#define FALSE 0
#define TRUE  1
#endif

#define GET_COMMAND_TEXT_PTR( nlm_handle_pptr ) \
   *((UINT8**) (nlm_handle_pptr) + 2)

#define PARSER_MAX_PARM_SIZE  31
#define PARSER_MAX_VALUE_SIZE 63 /* per value (element of list) */

#define PARSER_MAX_ADDRESS    20

#define NO_USER_HELP    NULL

#ifdef LITTLE_ENDIAN
#define STORE_BYTES( b1, b2, b3, b4 )  \
   (b4<<24) + (b3<<16) + (b2<<8) + b1
#else
#define STORE_BYTES( b1, b2, b3, b4 )  \
   (b1<<24) + (b2<<16) + (b3<<8) + b4
#endif

/*
 * Macros  to build the Parser command,  parameter and parameter value
 * tables.  The user_id is assigned by the user (0..239),  and will be
 * returned by  the  parser  as a way to identify the found parameter.
 * The (optional) help should be a single line of up to 55 characters,
 * preceeded by a 3 Tab characters  ("\t\t\t"),   and  terminated by a
 * NewLine character ("\n").
 */

#define PARSER_COMMAND_TABLE( command_table_name, command_id_string, \
                        command_flags, parameter_table_name,   \
                        parameter_preview_call_ptr )           \
static COMMAND_TABLE command_table_name =            \
{                                           \
   command_id_string, /* id, for error messages */ \
   command_flags,     /* PFC_... flags */         \
   parameter_table_name,                         \
   parameter_preview_call_ptr,                    \
};

#define PARSER_PARAMETER_TABLE( parameter_table_name ) \
static PARAMETER_TABLE parameter_table_name =          \
{
#define PARSER_PARAMETER_TABLE_END \
   { NULL },                   \
};
```

```
#define PTE_EQUATE( long_name, short_name, user_id, help, \
                parm_flags, equate_values )          \
    {                                        \
        long_name, short_name, help, user_id,            \
        PDE_EQUATE (parm_flags, equate_values)           \
    },
#define PTE_NAME( long_name, short_name, user_id, help, \
               parm_flags, min_size, max_size )      \
    {                                        \
        long_name, short_name, help, user_id,          \
        PDE_NAME (parm_flags, min_size, max_size)       \
    },
#define PTE_BOOLEAN( long_name, short_name, user_id, help, \
                 parm_flags )                        \
    {                                        \
        long_name, short_name, help, user_id,             \
        PDE_BOOLEAN (parm_flags)                         \
    },
#define PTE_INT_RANGE( long_name, short_name, user_id, help, \
                  parm_flags, range_values )          \
    {                                        \
        long_name, short_name, help, user_id,              \
        PDE_INT_RANGE (parm_flags, range_values)            \
    },
#define PTE_HEX_RANGE( long_name, short_name, user_id, help, \
                  parm_flags, range_values )          \
    {                                        \
        long_name, short_name, help, user_id,              \
        PDE_HEX_RANGE (parm_flags, range_values)            \
    },
#define PTE_INT( long_name, short_name, user_id, help, \
              parm_flags  )                       \
    {                                        \
        long_name, short_name, help, user_id,         \
        PDE_INT (parm_flags)                         \
    },
#define PTE_HEX( long_name, short_name, user_id, help, \
                parm_flags )                     \
    {                                        \
        long_name, short_name, help, user_id,         \
        PDE_HEX (parm_flags)                         \
    },
#define PTE_ADDRESS( long_name, short_name, user_id, help, \
                parm_flags, base, nr_bits, separator, \
                bytes_per_separator )              \
    {                                        \
        long_name, short_name, help, user_id,            \
        { PT_ADDRESS, parm_flags,                        \
          STORE_BYTES (base, nr_bits, separator,          \
                  bytes_per_separator) },           \
    },
#define PTE_COMPLEX_ADDRESS( long_name, short_name, user_id,    \
                    help, parm_flags, complex_addr_pvt ) \
    {                                               \
        long_name, short_name, help, user_id,                 \
        { PT_COMPLEX_ADDRESS, parm_flags|PFP_LIST_ALLOWED,      \
            complex_addr_pvt },                              \
    },
#define PTE_DUAL_TYPE( long_name, short_name, user_id, help, \
                    dual_type_pvt )                    \
    {                                              \
        long_name, short_name, help, user_id,              \
        { PT_DUAL_TYPE, 0, dual_type_pvt },                 \
    },
#define PTE_USER_PARSED( long_name, short_name, user_id, help, \
                    parm_flags )                        \
    {                                               \
        long_name, short_name, help, user_id,                 \
        { PT_USER_PARSED, parm_flags },                       \
    },

/*
 * The  following  macros  define parameter types that can be used for
 * the "dual type".
 */

#define PDE_EQUATE( parm_flags, equate_values ) \
        { PT_EQUATE, parm_flags, equate_values },

#define PDE_NAME( parm_flags, min_size, max_size ) \
        { PT_NAME, parm_flags|PFP_EMPTY_ALLOWED,   \
        STORE_BYTES (min_size, max_size, 0, 0) },

#define PDE_INT_RANGE( parm_flags, range_values ) \
        { PT_INT_RANGE, parm_flags, range_values },

#define PDE_HEX_RANGE( parm_flags, range_values ) \
        { PT_HEX_RANGE, parm_flags, range_values },

#define PDE_INT( parm_flags ) \
        { PT_INT_RANGE, parm_flags, NULL },

#define PDE_BOOLEAN( parm_flags ) \
        { PT_BOOLEAN, parm_flags|PFP_EMPTY_ALLOWED, NULL },

#define PDE_HEX( parm_flags ) \
        { PT_HEX_RANGE, parm_flags, NULL },

#define PDE_ADDRESS( parm_flags, base, nr_bits,       \
                separator, bytes_per_separator ) \
        { PT_ADDRESS, parm_flags, STORE_BYTES (base,  \
          nr_bits, separator, bytes_per_separator) },

/*
 * The following definitions are values passed to PTE_ADDRESS.
 */

#define HEX_ADDRESS     16 /* Address given in Hexadecimal */
#define DECIMAL_ADDRESS 10
#define OCTAL_ADDRESS    8

#define NO_SEPARATOR     0
#define BYTE_SEPARATOR   1 /* Separator between Bytes, */
#define WORD_SEPARATOR   2 /* Words (16 bits)*/
#define DWORD_SEPARATOR  4 /* or DoubleWords (32 bits) */

/*
 * Definitions for PTE_COMPLEX_ADDRESS (PVT initialization)
 */

#define PVE_COMPLEX_ADDRESS( base, nr_bits, separator,  \
                        bytes_per_separator )        \
    { base, nr_bits, separator, bytes_per_separator },
#define PVE_COMPLEX_ADDRESS_END { 0, 0, 0, 0 },

/*
 * In case an error,  after calling PARSER_GET_GIVEN_PARM_AND_VALUE or
 * PARSER_GET_NEXT_PARM_AND_VALUE the parser will return the following
 * USER_ID's.   Note  that  the parser will have issued an appropriate
 * error message.
 */

#define PUI_PARAMETER_MISSING 0xff /* required parameter is missing */
#define PUI_PARAMETER_UNKNOWN 0xfe /* unknown paramater */
#define PUI_VALUE_BAD         0xfd /* parameter value bad */
#define PUI_HELP_REQUEST      0xfc /* user has been shown help screen */

#define PUI_PARSER_DEFINED    0xf0 /* user has user_id's 0..239 */

/*
 * Definition of the command "flags"
 * ---------------------------------
 */

#define PFC_NONE                 0x0000
#define PFC_UNKNOWN_PARMS_ALLOWED 0x8000 /* allow (ignore) unknown parms */

/*
 * Definition of the parameter "flags"
 * -----------------------------------
 */

#define PFP_NONE           0x0000
#define PFP_REQUIRED       0x0001 /* Parameter (KW) required. Command */
                            /* fails when this parameter is omitted */
#define PFP_SUPPRESS_HELP  0x0002 /* suppress help for this parameter */
#define PFP_LIST_ALLOWED   0x0004 /* KW = (v1, v2, v3), or KW = v1 */
                            /* Lists can use either (..) or {..} */
```

```
#define PFP_DUPL_ALLOWED   0x0008 /* Allow same parameters more than once */
#define PFP_QUOTES_ALLOWED 0x0010 /* Parm value (name) may be quoted, */
                              /* allowing special characters */
#define PFP_PREVIEW_CALL   0x0020 /* Call preview procedure, user_id, */
                              /* parm_type, given_parm, given_value */
                              /* will be filled out, given value may */
                              /* be modified by the user */
#define PFP_EMPTY_ALLOWED  0x0200 /* Allow parameter without value */
                              /* e.g KW or KW="" */
#define PFP_UNSUPPORTED    0x0100 /* (currently) not supported parameter */
                              /* if specified, parser will give error */

/*
 * Parameter Types, and associated records (if applicable)
 * ------------------------------------------------------
 */
                        /* examples of usage: */
#define PT_USER_PARSED     0 /* parser does not process, user may */
#define PT_INT_RANGE       1 /* KW=17 or KW=0x1a (performs range checks) */
#define PT_HEX_RANGE       2 /* KW=1a0 or KW=0x1a0 (performs range checks) */
#define PT_BOOLEAN         3 /* KW=ON|OFF|TRUE|FALSE|YES|NO (or just KW) */
#define PT_NAME            4 /* KW=any_name (or KW = "any name") */
#define PT_EQUATE          5 /* KW=ieee|dec|none */
#define PT_ADDRESS         6 /* KW=8-0-02-1a-1-3b or 0800021a013b */
#define PT_COMPLEX_ADDRESS 7 /* KW=(5:a657:34:2330 1b-22-44 2) */

#define PT_DUAL_TYPE       9 /* KW=34 or KW=none */

typedef struct pv_int_range
{
   INT32   lower_value; /* lower value can be negative */
   INT32   upper_value;
   UINT8** dimension;   /* optional dimension (e.g. "Seconds") */

} PV_INT_RANGE;

#define PV_HEX_RANGE PV_INT_RANGE

typedef struct pv_equate_entry
{
   UINT8* name;  /* NULL is end of list */
   UINT16 value; /* value associated with the name */

} PV_EQUATE_ENTRY;

typedef PV_EQUATE_ENTRY  PV_EQUATE [];

typedef struct pv_name
{
   UINT8 min_length; /* minimum length (0..47) */
   UINT8 max_length; /* excluding string terminating zero (0..47) */

} PV_NAME;
```

```
typedef struct pv_address
{
   UINT8 base;               /* Base (8, 10, 16) */
   UINT8 bits;               /* maximum nr of bits (1..64) */
   UINT8 separator;          /* separator character ('-') */
   UINT8 separated_bytes;    /* between 0..4 bytes can be separated */

} PV_ADDRESS;

#define PV_COMPLEX_ADDRESS  PV_ADDRESS

/*
 * Parameter Value Table definition (overlapping)
 * ---------------------------------------------
 */

typedef struct pv_types
{
   union
   {
      void*              void_ptr;

      PV_EQUATE_ENTRY*    equ_tbl_ptr;
      struct pv_dual_type* dual_type_tbl_ptr;
      PV_INT_RANGE*       int_range_tbl_ptr;
      PV_COMPLEX_ADDRESS*  complex_address_tbl_ptr;
      PV_ADDRESS          address_tbl;
      PV_NAME             name_tbl;
   } u;

} PV_TYPES;

/*
 * Parameter list definitions
 * --------------------------
 */

typedef struct pv_parm_info
{
   UINT8     parm_type;
   UINT16    flags;
   PV_TYPES  pv_type;

} PV_PARM_INFO;

typedef struct pv_dual_type
{
   PV_PARM_INFO info_1;
   PV_PARM_INFO info_2;

} PV_DUAL_TYPE;
```

```
typedef struct parameter_entry
{
   UINT8*      name;          /* required (long or only) name */
   UINT8*      alternate_name; /* optional (short) name */
   UINT8**     user_help;     /* ptr->ptr allows for NLS support */
   UINT8       user_id;       /* assigned by caller (must be 0..239) */
   PV_PARM_INFO info;

} PARAMETER_ENTRY;

typedef PARAMETER_ENTRY PARAMETER_TABLE [];

/*
 * Parser interface record (defined by user, passed to parser)
 * ----------------------------------------------------------
 */

typedef void ERROR_CALL (char* error_text);

typedef struct parser_interface
{
   /* The following fields are initialized and used by the Parser only */

   UINT8              if_version;
   UINT8              spare1 [2];
   UINT8              list_index;
   struct command_table*  command_table_ptr;
   UINT8*              command_text_ptr;
   PARAMETER_ENTRY*      parm_list_entry_ptr;
   UINT8*              next_text_ptr;
   ERROR_CALL*          error_call_ptr;
   UINT8              get_next_count;

   /* The following fields can be referenced by the caller */

   UINT8      in_list;   /* non-0: KW=(v1,v2,v3), for v1 and v2 */
   UINT8      bad_value;
   UINT8      user_id;   /* copied from parm_list table */
   UINT8      given_parm  [PARSER_MAX_PARM_SIZE + 1];
   UINT8      given_value [PARSER_MAX_VALUE_SIZE + 1];
   short      parm_type; /* found parameter type (e.g. PT_INT_RANGE) */
   union
   {
      UINT32 uint;                    /* PT_INT/HEX_RANGE */
      INT32  sint;                    /* PT_INT/HEX_RANGE */
      UINT32 boolean;                 /* PT_BOOLEAN */
      UINT8  name [PARSER_MAX_VALUE_SIZE + 1]; /* PT_NAME */
      UINT32 equ;                     /* PT_EQUS */
      UINT8  address [PARSER_MAX_ADDRESS];    /* PT_ADDRESS */
      UINT8  user [PARSER_MAX_VALUE_SIZE + 1]; /* PT_USER_PARSED */

   } value;              /* returned value */
} PARSER_INTERFACE;

/*
 * Command parameter table definitions
 * ----------------------------------
 */

typedef void PARAMETER_PREVIEW_CALL (struct parser_interface* pi_ptr);

typedef struct command_table
{
   UINT8*                command_name;
   UINT32                command_flags;
   PARAMETER_TABLE*        parm_list_ptr;
   PARAMETER_PREVIEW_CALL* parm_preview_call_ptr;

} COMMAND_TABLE;

/*
 * ==================================
 * = Extern entry point definitions =
 * ==================================
 */

/* NAME:  P a r s e r I n i t i a l i z e
 *
 * FUNCTION:
 *     Initialize  the parser interface record.  At this time,  the parser
 *     will  also  check   for  errors  (using  only  the  table  provided
 *     information). When the user enters a '?' as a parameter, the parser
 *     will show the help screen.
 *
 * ALGORITHM:
 *     The  user  passes the command_table and the command_text addresses.
 *     This procedure  will  do  all  the  required  initialization of the
 *     parser interface record. A FALSE will be returned if any errors are
 *     found. The associated error message has already been displayed. The
 *     caller should not continue parsing when an error  is  detected,  as
 *     that may cause doubly displayed error messages. The given interface
 *     version must be less than or equal to the currently known  version,
 *     and no lower than the parsers allowed minimum (the parser knows the
 *     minimum level that is still compatible).
 *
 *     A  FALSE  will  also  be  returned  after the help display has been
 *     shown.  The user_id will reflect this (PUI_HELP_REQUEST).
 *
 *     The error_call_ptr allows the user to receive all the error message
 *     strings (for private logging purposes).
 *
 * PARAMETERS:
 *     parser_interface address
 *     command_table_ptr
 *     command_text_ptr
 *     if_version
 *     error_call_ptr (if NULL parser will call OutputToConsole)
 *
 * RETURN:
 *     TRUE - initialization is successful
 *     FALSE - error found (parsing with given table information)
 */

extern int ParserInitialize (PARSER_INTERFACE* pi_ptr,
                        COMMAND_TABLE* command_table_ptr,
                        unsigned char* command_text_ptr,
                        int if_version,
                        ERROR_CALL* error_call_ptr);


/* NAME:  ParserGetNextParameterAndValue
 *
 * FUNCTION:
 *     Obtain the next parameter and associated value for given command.
 *
 * ALGORITHM:
 *     This  procedure  locates  the next parameter in the command string.
 *     Parameters that have already  been  processed individually (through
 *     the use of ParserGetGivenParameterAndValue) will be skipped.
 *
 * PARAMETERS:
 *     Parser interface table
 *
 * RETURN:
 *     FALSE - end of command line or error (error message issued)
 *     TRUE - good parameter and value found
 */

extern int ParserGetNextParameterAndValue (PARSER_INTERFACE* pi_ptr);


/* NAME:  ParserGetGivenParameterAndValue
 *
 * FUNCTION:
 *     Get the value for a specifically given parameter.
 *
 * ALGORITHM:
 *     Locate  the  given  parameter  in the command text,  and return the
 *     value.  A specified parameter will  only  be returned to the caller
 *     once. This allows the caller to mix ParserGetGivenParameterAndValue
 *     and ParserGetNextParameterAndValue calls. This can be useful when a
 *     specific parameter is needed to determine if  a  default value must
 *     be used (when this parameter is omitted),  or when  parameters must
 *     be processed in a certain order,  other than entered on the command
 *     line.  A FALSE is returned when the parameter is not found, when it
 *     has already been seen by the user, or when it has a bad value.  The
 *     interface record will contain additional information in  case of an
 *     error.
 *
 * PARAMETERS:
 *      Parameter interface table
 *      Parameter name (either long or short form) - in UPPER case
 *
 * RETURN:
 *      FALSE - parameter not found, or has a bad value (see pi_ptr->..)
 *      TRUE - parameter and good value found
 */

extern int ParserGetGivenParameterAndValue (PARSER_INTERFACE* pi_ptr,
                                            UINT8* parm_name);


/* NAME:  ParserGetNextValue
 *
 * FUNCTION:
 *      Obtain the next parameter value for a list.
 *
 * ALGORITHM:
 *      Parameters  are  specified as <Keyw1 = v1>,  <Keyw2=v2> (spaces and
 *      commas are optional).  The user may only call this procedure when a
 *      list of parameters  <Keyw  =  (v1,v2,v3)>  is found,  to obtain the
 *      value of p2,  p3,  etc (the first parameter value  is passed by the
 *      ParserGetNextParameterAndValue  or  ParserGetGivenParameterAndValue
 *      call).
 *
 * PARAMETERS:
 *      pi_ptr
 *
 * RETURN:
 *      FALSE - error in value (message issued)
 *      TRUE - good value found for current parameter
 */

extern int ParserGetNextValue (PARSER_INTERFACE* pi_ptr);

/*
 * Export of some general purpose string manipulation procedures.  The
 * Parser will attempt to import NLS aware symbols. and use those when
 * available. This eliminates the forced dependency on CLIB.
 *
 * Note  that  ParserXprintf  only  supports  the  very  basic  string
 * replacements. They are: %x, %d, %0Nx (N=1..9), %c, %s and %%.
 */

typedef char *VA_LIST [1];
extern int ParserSprintf (char*, char*, ...);
extern int ParserVsprintf (char *s, const char *format, VA_LIST arg);
extern char* ParserIncrement (char*, int);
extern char ParserCharVal (char*);
extern char ParserCharUpr (char);
extern int ParserStrlen (char*);
extern char* ParserStrcat (char* dst, const char* t);
extern char* ParserStrcpy (char* dst, const char* t);
extern char* ParserUtoa (unsigned value, char* buffer, unsigned int radix);

#endif
```

PARSER.IMP

```
#
# (C) Copyright 1995 Novell, Inc., All Rights Reserved.
#
# File Name       : parser.imp
# Author          : Henk J. Bots
# Created On      : Thu, Sep 21 (11:35:17) 1995
# Last Modified By: Henk J. Bots
# Last Modified On: Wed, Nov 29 (14:19:43) 1995
# Update Count    : 4
# Status          : Parser Exported Symbols.
#
  ParserInitialize,
  ParserGetNextValue,
  ParserGetNextParameterAndValue,
  ParserGetGivenParameterAndValue,
  ParserSprintf,
  ParserVsprintf,
  ParserIncrement,
  ParserCharVal,
  ParserCharUpr,
  ParserStrlen,
  ParserStrcat,
  ParserStrcpy,
  ParserUtoa
```

appendix A Glossary

ADS

Adapter Data Space.

API

Application Program Interface.

ATM

Asynchronous Transfer Mode. A cell-switching communication technology that supports data, voice and video. Speeds vary from the low Mbits to Gbits.

ATM Forum

An industry alliance of more than 500 companies dedicated to rapidly standardizing ATM through design and specification work.

ATM Switch

A hardware device that takes an incoming ATM cell and directs it to one or more of many potential output interfaces.

CO-IPX

Connection Oriented IPX.

ECB

Event Control Block: a NetWare data buffer, also containing qualifying information about the data.

Gbits

Giga bits per second (equivalent to 1,000 Mbits).

HSM

Hardware Specific Module. The lowest component of the ODI architecture, responsible for the hardware specific handling of an adapter boards. Also called the driver.

ILMI

Interim Local Management Interface.

ISDN

Integrated Services Digital Network.

ITU

International Telephone Union.

LLC

Logical Link Control.

LSL

Link Support Layer.

LMI

Layer Management Interface.

MAC

Media Access Control.

Mbits

Mega bits per second (One million bits per second).

MLID

Multiple Link Interface Driver.

MSM

Media Support Module.

NLM

NetWare Loadable Module.

NNI

Network-to-Network Interface.

ODI

Open Data-link Interface.

PVC

Permanent Virtual Connection.

QoS

Quality of Service. Term to describe delay, throughput, bandwidth, and so on of a Virtual Connection.

SDU

Service Data Unit

SVC

Switched Virtual Connection.

TSM

More correctly referred to as the ATMTSM or the Topology Specific Module. A component of the ODI architecture that is specific to the media-type (such as Ethernet, Token-Ring, FDDI, etc.).

UNI

User-to-Network Interface.

VC

Virtual Connection. A point-to-point or point-to-multipoint connection between ATM end stations. Can either be switched or permanent.

VCI

Virtual Connection Identifier. A 32-bit identifier, consisting of the 16bit Virtual Path Connection Identifier and the 16-bit Virtual Channel Identifier.

Novell®
Everything's Connected™

appendix B References

ATM User-Network Interface Specification Version 3.0

ITU Draft Recommendation Q.SAAL1, DT/11/3-38

NetWare ODI Server Driver Specification

Novell®
Everything's Connected™

Index