MMS-EASE

Reference Manual

Revision 11

Systems Integration Specialists Company, Inc.
6605 19½ Mile Road, Sterling Heights, MI 48314, U.S.A.
Tel: (810) 254-0020, Fax: (810) 254-0053
E-Mail: support@sisconet.com, URL: http://www.sisconet.com

Printed in U.S.A.

10/97

DWG # 100064/1.5

MMS-EASE

Reference Manual

Revision 11

10/31/97

1. Introduction

The SISCO **MMS E**mbedded **A**pplication **S**ervice **E**lement (MMS-EASE) consists of a library of C functions that implement a high-level, real-time **A**pplication **P**rogram **I**nterface (API) to the **M**anufacturing **M**essage **S**pecification (MMS). The MMS-EASE API consists of two parts, the **P**aired-**P**rimitive **I**nterface (PPI) and the **V**irtual **M**achine **I**nterface (VMI).

The Paired-Primitive Interface of MMS-EASE is in a form that closely parallels the MMS Service Primitives specified in the MMS Protocol of ISO 9506. However, it goes far beyond what is normally a primitive-level interface.

The Virtual Machine Interface provides a higher-level of functionality than the paired-primitive interface by providing automatic functions to take care of actions specific to the operating system such as reading and writing variables and files. In addition, the virtual machine interface can group requests and responses transparently to the user so it is simple to deal with issues like reading multiple file segments.

MMS-EASE is embedded in the application, in the sense that by linking in with the user program, it becomes indivisible from it. Below is a diagram depicting the organization of MMS-EASE applications:

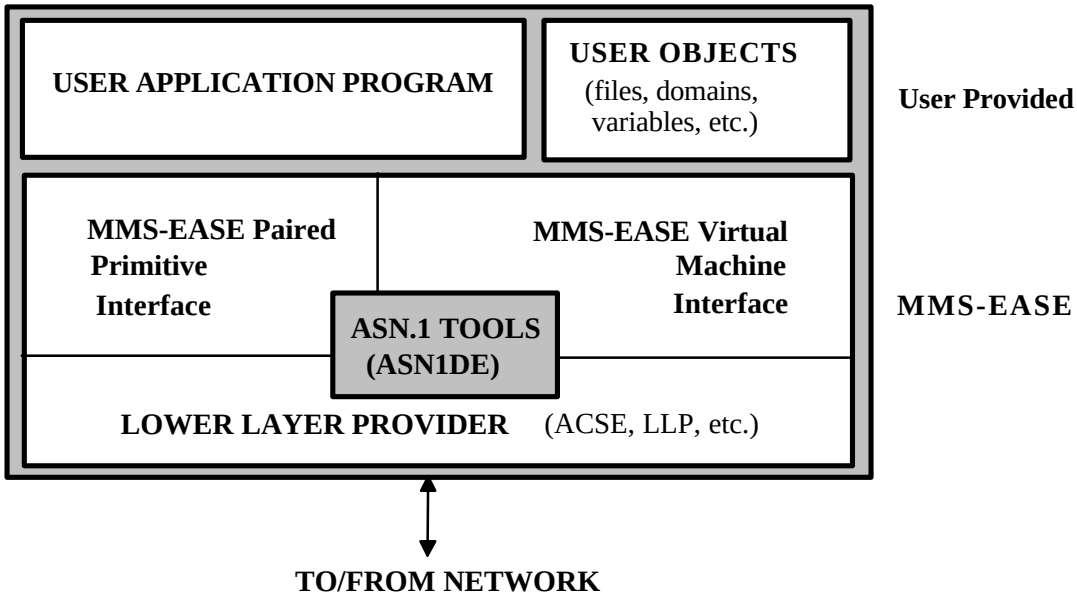


FIGURE 1: Major Components of MMS-EASE

MMS-EASE is based on and includes SISCO's **ASN.1 D**ecoder/**E**ncoder (ASN1DE). This consists of general utility functions that allow efficient building and parsing of MMS (and other ASN.1) messages. Also included is SUIC, that provides an easy-to-use and consistent interface to the ACSE level functions of the particular interface board or OSI implementation used.

As Figure 1 shows, MMS-EASE is written in modular fashion, with each part using the services of the others. The vast majority of applications do not need to deal with these other interfaces even though most of these internal interfaces are exposed. The MMS-EASE libraries directly provide almost everything needed by a MMS application. The user writes the application program, compiles it, and then links the user object code with the MMS-EASE libraries to create an executable MMS compatible application.

Prerequisites

Because of the technical nature of MMS-EASE, OSI, and MMS, some level of knowledge is required to fully understand how to use MMS-EASE. The required areas of knowledge are:

1. Familiarity with OSI and MMS specifications particularly MMS: ISO IS 9506. We strongly suggest purchasing the MMS specification. Information can be obtained from the following sources:

   SPECIFICATIONS:
   ANSI (**A**merican **N**ational **S**tandards **I**nstitute)
   1430 Broadway
   New York, NY 10018

   ISO (International Organization for Standardization)
   1 Rue de Varenbe
   Case Pascal 66 CH-1211
   Geneva 20 Switzerland

   A portion of the ISO IS 9506-2 document is available on MMS-EASE products that are supplied on CD.

2. There must be a strong knowledge of the ANSI C programming language. This is also a requirement.

Training

SISCO's MMS-EASE training programs are designed to provide MMS-EASE training on OSI, MMS, and MMS-EASE in a modular fashion. This is provided so you can pick the training suitable to your own needs. All training classes are conducted by experienced professionals who are familiar not only with MMS-EASE, but with OSI and MMS technology in general.

Two courses are currently available:

1. MMS: A Practical Approach — This two-day workshop program provides an overview to MMS that is non-commercial and not specific to a particular MMS product. The classroom portion of the course provides detailed information on the justification and application of MMS. The workshop portion includes hands-on projects where students work in a team to determine MMS compatibility requirements and design the MMS messaging needed to implement a simple manufacturing system. This course is intended for systems designers and project managers.

2. MMS-EASE Implementor's Workshop — This is a five-day program which will provide you with instruction in the use of MMS-EASE, SISCO's Application Programming Interface (API) for MMS Protocols. This course is designed for programmers and engineers who will be actively designing, programming, or testing applications for systems requiring MMS-EASE. Course emphasis is placed on building familiarity with the MMS-EASE environment through hands-on programming laboratory assignments. To attend this course, you must have programming experience using the 'C' language. Some knowledge of MMS is helpful but not required.

Contact SISCO for further information on the availability of these and other training courses.

Telephone:	~~810-~~254-0020
Fax:	~~810-~~254-0053
Support and Training Email:	support@sisconet.com
General Information Email:	info@sisconet.com
URL:	http://www.sisconet.com
FTP:	ftp.sisconet.com login as anonymous

How To Use This Manual

There are three volumes to the MMS-EASE Reference Manual. Within these volumes, modules are used to group topics separated by tabs.

VOLUME 1

- **INTRODUCTION — MODULE 1**

  Overview of MMS and Addressing — provides an overview of MMS, a description of MMS services, and network addressing issues in OSI, TCP/IP, and Directory Services.

  Introduction to MMS-EASE — provides descriptions of Paired Primitive and Virtual Machine Interfaces, and Function Classes.

- **GENERAL — MODULE 2**

  General Description of MMS-EASE — provides descriptions of the general sequence of events for confirmed and unconfirmed services, Request and Indication control, and Data Structures.

  MMS-EASE Programming Reference — provides information specific to the application programmer such as configuration, debug facilities, subset creation, linked lists, memory management, operation-specific data structures, general interface functions, and utility tools.

- **CONTEXT MANAGEMENT — MODULE 3**

  This section provides information on the Context Management services, including the data structures, and functions.

VOLUME 2

- **VMD SUPPORT — MODULE 4**

  This provides information on the VMD Support services, including the data structures, and functions.

- **VARIABLE ACCESS — MODULE 5**

  This provides information on the Variable Access and Management services, including the data structures, and functions. It also explains SISCO's **TDL** (**T**ype **D**escription **L**anguage), runtime type and alternate access functionality and use of SISCO's **ADL** (**A**lternateAccess **D**escription **L**anguage).

- **DOMAINS AND PROGRAM INVOCATIONS — MODULE 6**

  Domain Management — provides information on the Domain Management services, including the data structures and functions.

  Program Invocation Management — provides information on the Program Invocation Management services, including the data structures and functions.

VOLUME 3

- **SEMAPHORES, EVENTS, AND JOURNALS — MODULE 7**

  Semaphore Management — provides information on the Semaphore Management services, including the data structures and functions.

  Event and Alarm Management — provides information on the Event and Alarm Management services, including the data structures and functions.

  Journal Management — provides information on the Journal Management services, including the data structures and functions.

- **OPERATOR COMMUNICATION — MODULE 8**

  This provides information on the Operator Communication services, including the data structures and functions.

- **FILE ACCESS AND MANAGEMENT — MODULE 9**

  This provides information on the File Access and Management services, including the data structures and functions.

- **THIRD PARTY HANDLING — MODULE 10**

  This provides information on Third Party Handling, including the data structures and functions.

- **ERROR HANDLING — MODULE 11**

  This provides information on Error Handling, including the data structures, and functions.

In addition, Volume 1 contains the following appendices:

- **APPENDIX A — Error Codes —** provides information on error and reject codes used by MMS-EASE.
- **APPENDIX B — Opcodes for User-Defined Functions —** provides a list of operation codes used for **u_xxxx_ind** and **u_mp_xxxx_conf** functions.
- **APPENDIX C — File Descriptions —** provides a list of file descriptions provided with the MMS-EASE product.
- **APPENDIX D — Service vs. Function Naming Conventions —** provides the abbreviations that are designated "**xxxx**" for various data structures and functions related to a given MMS service.
- **APPENDIX E — Global Data Type Definitions —** provides an explanation of the variable type definitions pre-defined in file, **glbtypes.h**.
- **APPENDIX F — MMS-EASE Demonstration Program —** provides an explanation of the MMS-EASE demo program. This includes configuration and how to create it.

Separate table of contents and indexes are contained in each volume along with a master table of contents and index found in Volume 1. There is also a place to file installation guides, release notes, and other supplemental materials associated with your MMS-EASE product.

Conventions Used In This Manual

Function names, structures, and members of functions and structures are shown in **`boldface courier`** type.

Code fragments are shown in `Courier`.

2. Overview of MMS

Manufacturing **M**essage **S**pecification (MMS) is an internationally standardized messaging system for exchanging real-time data and supervisory control information between networked devices and/or computer applications in a manner that is independent of: 1) the application function being performed or 2) the developer of the device or application. MMS is an international standard (ISO 9506) developed and maintained by Technical Committee Number 184 (TC184), Industrial Automation, of the **I**nternational **O**rganization for **S**tandardization (ISO).

The messaging services provided by MMS are generic enough to be appropriate for a wide variety of devices, applications, and industries. For instance, the MMS Read service allows an application or device to read a variable from another application or device. Whether the device is a **P**rogrammable **L**ogic **C**ontroller (PLC) or a robot, the MMS services and messages are identical. Similarly, applications as diverse as material handling, fault annunciation, energy management, electrical power distribution control, inventory control, and deep space antenna positioning in industries as varied as automotive, aerospace, petro-chemical, electric utility, office machinery and space exploration have put MMS to useful work.

The History of MMS

In the early 1980s, a group of **N**umerical **C**ontroller (NC) vendors, machine builders and users working under the auspices of committee IE31 of the **E**lectronic **I**ndustries **A**ssociation (EIA) developed a draft standard proposal #1393A, entitled "User Level Format and Protocol for Bidirectional Transfer of Digitally Encoded Information in a Manufacturing Environment." When the General Motors Corporation began its **M**anufacturing **A**utomation **P**rotocol (MAP) effort in 1980 they used the EIA-1393A draft standard proposal as the basis for a more generic messaging protocol. It could be used for NCs, **P**rogrammable **L**ogic **C**ontrollers (PLCs), robots and other intelligent devices commonly used in manufacturing environments. The result was the **M**anufacturing **M**essage **F**ormat **S**tandard (MMFS). MMFS was used in the MAP Version 2 specifications published in 1984. During the initial usage of MMFS it became apparent that a more rigorous messaging standard was needed. MMFS allowed too many choices for device and application developers. This resulted in several mostly incompatible dialects of MMFS. Furthermore, MMFS did not provide sufficient functionality to be useful for the **P**rocess **C**ontrol **S**ystems (PCS) found in continuous processing industries. With the objective of developing a generic and non-industry specific messaging system for communications between intelligent manufacturing devices the MMS effort was begun under the auspices of Technical Committee Number 184, Industrial Automation, of the International Organization for Standardization (ISO).

The result was a standard based upon the **O**pen **S**ystems **I**nterconnection (OSI) networking model called the Manufacturing Message Specification (MMS). A **D**raft **I**nternational **S**tandard (DIS) version of MMS was published in December 1986 as ISO DIS 9506. The DIS version of MMS (Version 0) overcame the problems with MMFS but had not yet advanced to the status of an **I**nternational **S**tandard (IS). Faced with a publication deadline of November 1988 the MAP technical committees referenced the DIS version of MMS for the MAP V3.0 specification. In December 1988 the IS version of MMS (Version 1) was released as ISO 9506 parts 1 and 2. It was not until after the development of backwards compatibility agreements by the **N**ational **I**nstitute of **S**tandards and **T**echnology (NIST) that the IS version of MMS was referenced by the MAP V3.0 specifications[1].

The MMS Standard

The MMS standard (ISO 9506), jointly managed by Technical Committee Number 184, Industrial Automation, of ISO and the International Electrotechnical Commission (IEC), consists of two or more parts (see "Companion Standards" on page 1-12). Parts 1 and 2 define what is referred to as the "core" of MMS. Part 1 is the *service* specification. The service specification contains a definition of 1) the Virtual Manufacturing Device, 2) the services (or messages) exchanged between nodes on a network, and 3) the attributes and

[1] Backwards compatibility was a requirement for any changes to the MAP V3.0 specification due to the 6-year stability commitment made by the MAP Steering Committee in 1988.

parameters associated with the VMD and services. Part 2 is the *protocol* specification. The protocol specification defines the rules of communication that includes 1) the sequencing of messages across the network, 2) the format (or encoding) of the messages, and 3) the interaction of the MMS layer with the other layers of the OSI model. The protocol specification uses a presentation layer standard called the **A**bstract **S**yntax **N**otation Number **One** (ASN.1 — ISO 8824) to specify the format of the MMS messages.

MMS provides a rich set of services for peer-to-peer real-time communications over a network. MMS has been used as a communication protocol for many common industrial control devices like CNCs, PLCs, robots. There are MMS applications in the electrical utility industry such as in **R**emote **T**erminal **U**nits (RTUs), **E**nergy **M**anagement **S**ystems (EMS) and other **I**ntelligent **E**lectronic **D**evices (IEDs) like reclosers and switches. Most popular computing platforms have MMS connectivity available either from the computer manufacturer or by a third party. Some of the computer applications available include **A**pplication **P**rogramming **I**nterfaces (APIs), **A**pplication **E**nablers (A/Es), **R**elational **D**ata **B**ase **M**anagement **S**ystems (RDBMS), and graphical monitoring systems, gateways, and drivers for spreadsheets, word processors. MMS implementations support a variety of communications links including Ethernet, Token Bus, RS-232C, OSI, TCP/IP, MiniMAP, FAIS, etc., and can connect to many more types of systems using networking bridges, routers, and gateways.

Benefits of MMS

MMS provides benefits by lowering the cost of building and using automated systems. In particular, MMS is appropriate for any application requiring a common communications mechanism for performing a diversity of communications functions related to real-time access and distribution of process data and supervisory control. When looking at how the use of a common communications service like MMS can benefit a particular system, it is important to evaluate the three major effects of using MMS that can contribute to cost savings: 1) Interoperability, 2) Independence and 3) Access.

- **Interoperability** is the ability of two or more networked applications to exchange useful supervisory control and process data information between them without the user of the applications having to create the communications environment. While many communication protocols can provide some level of interoperability, many of them are either too specific (to brand/type of application or device, network connectivity, or function performed — see Independence below) or not specific enough (provide too many choices for how a developer uses the network).

- **Independence** allows interoperability to be achieved independent of:

  •*The Developer of the Application*. Other communications schemes are usually specific to a particular brand (or even model in some cases) of application or device. MMS is defined by independent international standards bodies with participation from many leading industry experts and vendors.

  •*Network Connectivity*. MMS becomes *the* interface to the network for applications thereby isolating the application from most of the non-MMS aspects of the network and how the network transfers messages from one node to another.

  •*Function Performed*. MMS provides a common communications environment independent of the function performed. An inventory control application accesses production data contained in a control device in the exact same manner as an energy management system would read energy consumption data from the same device.

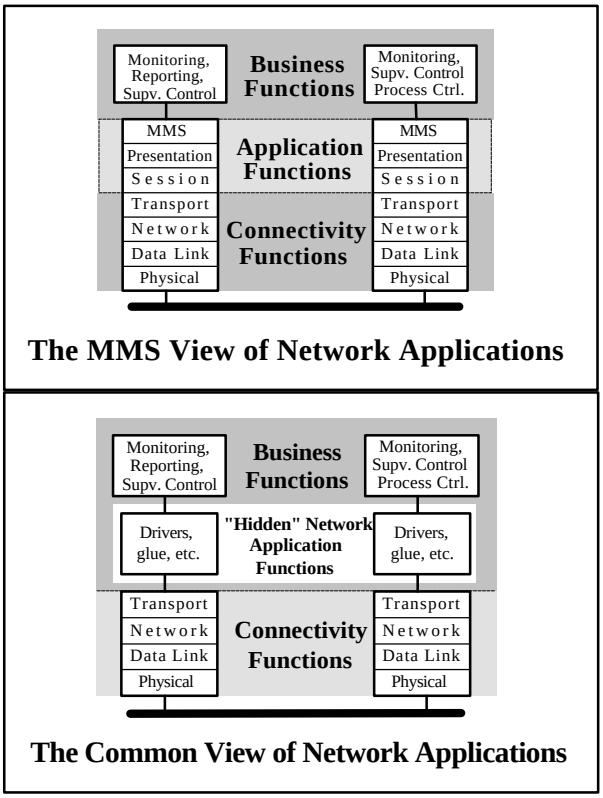
- **Data Access** is the ability of networked applications to obtain the information required by an application to provide a useful function. Although virtually any communications scheme can provide access to data at least in some minimal manner, they lack the other benefits of MMS particularly Independence (see above).

MMS is rigorous enough to minimize the differences between applications performing similar or complimentary functions while still being generic enough for many different kinds of applications and devices. Communications schemes that are not specific enough can result in applications that all perform similar or complementary functions in different ways. The result is applications that cannot communicate with each other because the developers all made different choices when implementing. While many communications schemes only provide a mechanism for transmitting a sequence of bytes (a message) across a network, MMS does much more. MMS also provides definition, structure and meaning to the messages that significantly enhances the likelihood of two independently developed applications interoperating. MMS has a rich set of

features that facilitate the real-time distribution of data and supervisory control functions across a network in a client/server environment that can be as simple or sophisticated as the application warrants.

Justifying MMS

The real challenge in trying to develop a business justification for MMS (or any network investment) is in assigning value to the benefits given a specific business goal. To do this properly it is important in understanding the relationship between the application functions, the connectivity functions and the business functions of the network (see figure below). In some cases the benefit of the common communications infrastructure MMS provides is only realized as the system is used, maintained, modified, and expanded over time. Therefore a justification for such a system must look at *life cycle costs* versus just the purchase price.



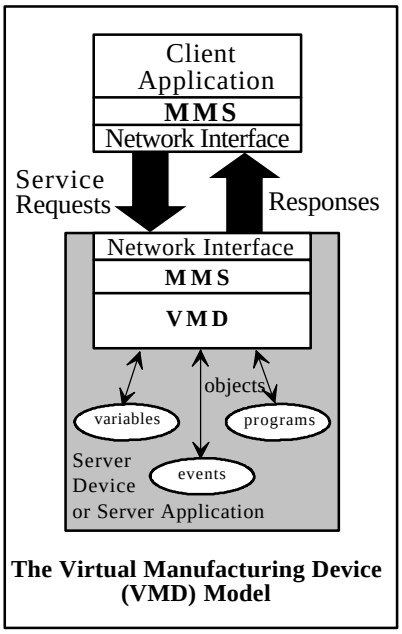
In order to assign benefit to the use of MMS it is important to first understand the value that MMS provides to applications. MMS, as an application layer protocol, provides application services to the business functions, not connectivity services. It is common to view the network simply as a mechanism to transfer messages (connectivity only). That view hides the value of the application functions because they become indistinguishable from the business applications. However, the costs are still there. Justifying MMS requires that the user recognize the value provided by the network application functions in facilitating interoperability, independence and access to data.

Figure 2: MMS Vs Common View of Network Applications

It is also important not to underestimate the cost associated with developing, maintaining, and expanding the application functions created if MMS is not used. A key element in assigning value is understanding that the business functions are what provide value to the enterprise. The cost of the custom network application functions directly reduces the amount of effort (i.e., cost) placed on developing, maintaining, and expanding the business functions. With MMS, the communications infrastructure is built once and then re-used by all the business functions.

MMS Application Services

The key feature of MMS is the Virtual Manufacturing Device (VMD) model. The VMD model specifies how MMS devices, also called servers, behave as viewed from an external MMS client application point of view. MMS allows any application or device to provide both client and server functions simultaneously.



The VMD model provides a consistent and well defined view to client applications of the objects contained in the VMD. Clients use MMS services to access and manipulate those objects. MMS requires that all servers behave according to the VMD model.

Figure 3: VMD Model

In general the VMD model defines:

1. objects (e.g., variables) that are contained in the server,
2. the services that a client can use to access and manipulate these objects (e.g., read or write a variable), and
3. the behavior of the server upon receipt of these service requests from clients.

The remainder of this overview of MMS provides a summary of the objects defined by the VMD model and the MMS services provided to access and manipulate those objects. While the range of objects and services is broad, a given device or application need only implement whatever subset of these objects and services that are useful in that situation.

A more detailed discussion of the MMS VMD model and the various MMS objects and their services are presented in the following section.

VMD Object

The VMD itself can be viewed as an object to which all other MMS objects are subordinate (such as variables, and domains contained within the VMD). MMS provides services such as Status, UnsolicitedStatus, and Identify for obtaining information and status about the VMD. It also provides services as GetNameList and Rename for managing and obtaining information about objects defined in the VMD.

Variable and Type Objects

MMS provides a comprehensive and flexible framework for exchanging variable information over a network. The MMS variable access model includes capabilities for named, unnamed (addressed), and named lists of variables. MMS also allows the type description of the variables to be manipulated as a separate MMS object (named type object). MMS variables can be simple (e.g., integer, boolean, floating point, string) or complex (e.g., arrays and structures).

The services available to access and manage MMS variable and type objects are very powerful and include services for:

- Read and Write services allow MMS client applications to access the contents of Named Variables, Unnamed Variables, and Named Variable List objects.
- The InformationReport service allows a server to report the contents of a variable to a remote client in an unsolicited manner.
- Define, delete, and get attribute services are available for both variables and types to allow clients to manage the variable access environment.
- Service options allow groups of variables to be accessed in a single MMS request, allow large arrays and complex structures to be partially accessed (alternate access), and allow clients to recast variable types and complex structures to suit their needs (access by description).

Program Control Objects (Domains and Program Invocations)

The VMD execution model defines two objects for controlling the execution of programs within the VMD. A MMS *domain* is defined as an object representing a resource within the VMD (e.g., the memory in which a program is stored). A *program invocation* is defined as a group of domains the execution of which can be controlled and monitored. Some of the features of the services the VMD execution model provides for MMS clients are:

- services for commanding a VMD to upload/download their domains to/from a MMS client or file in a filestore system (either in the VMD or external to the VMD),
- services for a VMD to request a domain upload/download from a client,
- Start, Stop, Reset, Resume, and Kill services for controlling the execution of program invocations,
- services for deleting, creating, and obtaining the attributes of domains and program invocations, and
- state changes in program invocations that can be linked to MMS events.

Event Objects

The MMS event management model defines several named objects consisting of:

1. **event conditions:** an object that represents the state of an event (i.e., active or idle),
2. **event actions:** an object that consists of the action that should be taken by the VMD upon the occurrence of a change in state of the event condition, and
3. **event enrollments:** an object that represents which MMS clients should be notified upon a change in state of an event condition.

The event management model provides a rich set of services for MMS clients:

- services for notifying clients about events and for clients to acknowledge these notifications,
- services for obtaining summaries of event conditions and event enrollments (called alarm summaries), and
- services for deleting, defining, obtaining the status and attributes of, and controlling event conditions, event actions, and event enrollments.

Semaphore Objects

MMS semaphores are named objects used to control access to other resources and objects within the VMD. For instance, a VMD that controls access to a setpoint (a variable) for a control loop could use semaphores to only allow one client at a time to be able to change the setpoint (e.g., with the MMS Write service). The MMS semaphore model defines two kinds of semaphores. *Token semaphores* represent a specific resource within the control of the VMD. *Pool semaphores* consist of one or more *named tokens* each representing a similar but distinct resource under the control of the VMD. MMS provides semaphore services allow MMS clients to:

- Take control of and relinquish the control of semaphores.
- Define, delete, and get the attributes or status of semaphores.

Journal Objects

A MMS journal is a named object representing a time based record, or log of data. Each entry in a journal can contain the state of an event, the value of a variable, or character string data (called *annotation*) that the VMD, or a MMS client, enters into the journal. Services available allow journals to created, read, deleted, and cleared (in whole or in part).

Operator Station Object

The operator station is an object representing a means of communicating with the operator of the VMD through a keyboard and display. An Output service is available to display an alpha-numeric string on a text display. An Input service is available to obtain alpha-numeric keyboard input with and without a text prompt on the display.

Files

An annex of MMS provides a simple set of services for transferring, renaming, and deleting files in a VMD. A FileDirectory service is provided to obtain a list of available files.

The Virtual Manufacturing Device (VMD) Model

The primary goal of MMS was to specify a standard communications mechanism for devices and computer applications that would achieve a high level of interoperability. To achieve this goal it would be necessary for MMS to define much more than just the format of the messages to be exchanged — a common message format, or protocol, is only one aspect of achieving interoperability. In addition to protocol, the MMS standard also provides definitions for:

Objects. MMS defines a set of common objects (e.g., variables, programs, events) and defines the network visible attributes of those objects (e.g., name, value, type).

Services. MMS defines a set of communications services (e.g., read, write, delete) for accessing and managing these objects in a networked environment.

Behavior. MMS defines the network visible behavior that a device should exhibit when processing these services.

This definition of objects, services, and behavior comprises a comprehensive definition of how devices and applications communicate that MMS calls the **V**irtual **M**anufacturing **D**evice (VMD) model. The VMD model only specifies the network visible aspects of communications. The internal detail of how a real device implements the VMD model (i.e., the programming language, operating system, CPU type, **I**nput/**O**utput (I/O) systems) are not specified by MMS. By focusing only on the network visible aspects of a device, the VMD model is specific enough to provide a high level of interoperability while still being general enough to allow innovation in application/device implementation and making MMS suitable for application across a range of industries and device types.

Client/Server Relationship

A key aspect of the VMD model is the client/server relationship between networked applications and/or devices. A server is a device or application that contains a VMD and its objects (variables, etc.). A client is a networked application (or device) that asks for data or an action from the server. In a very general sense, a client is a network entity that issues MMS service requests to a server. A server is a network entity that responds to the MMS requests of a client (see figure below). While MMS defines the services for both clients and servers, the VMD model defines the network visible behavior of servers only.

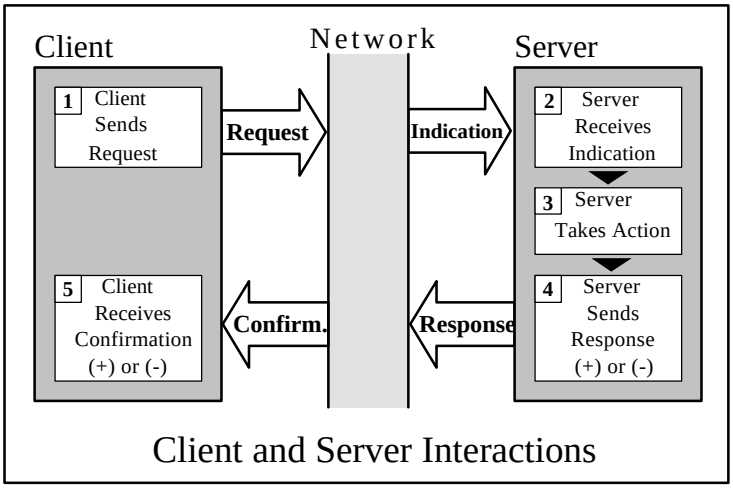


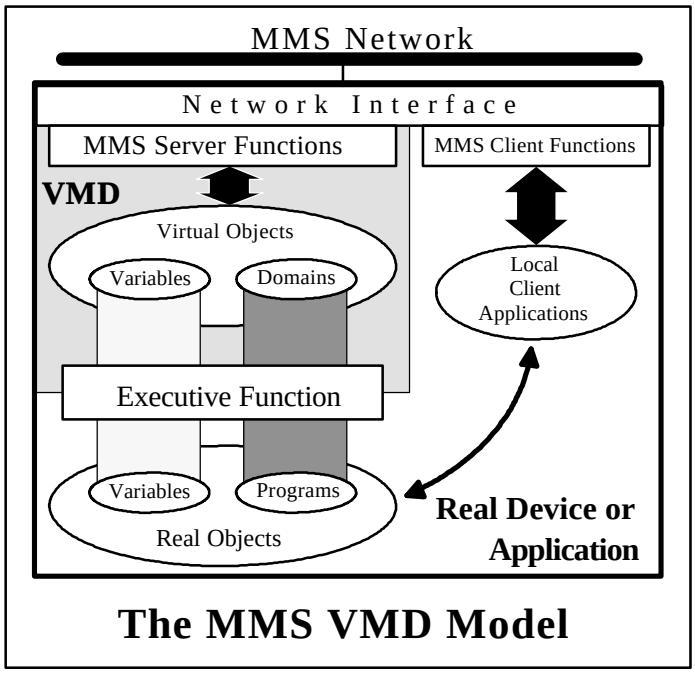
Figure 4: Client and Server Interactions

Client/Server Interactions. *MMS Clients and servers interact with each other by sending/receiving request, indication, response, and confirmation Service Primitives over the network. The diagram above depicts the interactions between a client and server for a MMS Confirmed Service where 1) the client sends a request, 2) the server receives an indication, 3) the server performs the desired action, 4) the server sends a positive (+) response if the action was successful or a negative (-) response if there was an error, and 5) the client receives the confirmation (+) or (-). An Unconfirmed Service is sent by the server and has only the request and indication service primitives (see InformationReport, UnsolicitedStatus, and EventNotification for examples of unconfirmed services)*

Many MMS applications and MMS compatible devices provide both MMS client and server functions. The VMD model would only define the behavior of the server functions of those applications. Any MMS application or device that provides MMS server functions must follow the VMD model for all the network visible aspects of the server application or device. MMS clients are only required to conform to rules governing message format or construction and sequencing of messages (the protocol).

"Real" and "Virtual" Devices and Objects

There is a distinction between a real device (e.g., a PLC, CNC, or robot) and the real objects contained in it (e.g., variables, programs) and the virtual device and objects defined by the VMD model. Real devices and objects have peculiarities (a.k.a. product features) associated with them that are unique to each brand of device or application. Virtual devices and objects conform to the VMD model and are independent of brand, language, operating system, etc. Each developer of a MMS server device or MMS server application is responsible for "hiding" the details of their real devices and objects, by providing an executive function. The executive function translates the real devices and objects into the virtual ones defined by the VMD model when communicating with MMS client applications and devices. Because MMS clients always interact with the virtual device and objects defined by the VMD model, the client applications are isolated from the specifics of the real devices and objects. A properly designed MMS client application can communicate with many different brands and types of devices in the same manner because the details of the real devices and objects are hidden from the MMS client by the executive function in each VMD. This virtual approach to describing server behavior does not constrain the development of innovative devices and product features and improvements. The MMS VMD model places constraints only on the network visible aspects of the virtual devices and objects, not the real ones.



Real and Virtual Objects. *The executive function provides a translation, or "mapping" between the MMS defined virtual objects and the real device objects used by the real device. Applications local to the VMD, and the objects contained in them, are only accessible to a remote MMS client application if the executive function provides the mapping function for those objects and applications. Client applications local to the VMD may access and manipulate the real objects without using MMS.*

Figure 5: MMS VMD Model

MMS Device and Object Modeling

The implementor of the executive function (the application or device developer) must decide how to "model" the real objects as virtual objects. The manner in which these objects are modeled is critical to achieving interoperability between clients and servers among many different developers. Inappropriate or incorrect modeling can lead to an implementation that is difficult to use or difficult to interoperate with.

For instance, take the situation of a PLC that contains a ladder program (a real object). The MMS implementor (the designer of the executive function) wishes to allow external client applications to copy the program from the PLC to another computer. For the purposes of this example we shall assume that the MMS VMD model gives the implementor the choice of modeling the ladder program object as a variable or domain object (both virtual objects). The choice of which virtual object to map to the real ladder program object is critical because MMS provides a set of services to manipulate variables that are quite different from the services used to manipulate domains. Variables can be read individually or in a list of typed data. Domains can be copied in whole only. If the ladder program is modeled as a MMS variable, it makes the task of performing a simple copying of the program complex. This is due to the nature of the ladder program data (typically a large block of contiguous memory) resulting in an extremely large variable that would be difficult to access easily. If the ladder program is modeled as a domain, there are specific MMS services provided for uploading and downloading the large blocks of untyped data typical of ladder programs. An incorrect modeling choice can make the real object difficult to access.

In some cases it makes sense to represent the same real object with two different MMS objects. For instance, a large block of variables may also be modeled as a domain. This would provide the MMS client the choice of services to use when accessing the data. The variable services would give access to the individual elements in the block. The domain services would allow the entire block to be read/uploaded or written/downloaded as an element of a program invocation (see the batch controller example on page 1-17).

Companion Standards

In many cases the relationship between the real objects and the virtual objects can be standardized for a particular class of device or application (e.g., a PLC or PCS). Developers and users of these real devices can define more precisely how MMS is applied to a particular class of device or application. The result is a *companion standard*.

In general, companion standards perform the following functions:

- Define the mapping of real objects to the VMD model for a particular class of device. For instance, the companion standard for process control systems would define the relationships between real objects such as setpoints, alarm limits and **P**roportional-**I**ntegral-**D**erivative (PID) control loops to MMS domains, variables, and program invocations.
- Provide additional definition of the behavioral characteristics of the VMD model for particular classes of devices. For instance, the actions a PLC takes when receiving a MMS Start or Stop service request are very different from the actions taken by robots to these service requests.
- Define additional network visible attributes for common objects. For example, the PLC companion standard defines additional status information for use in the MMS Status and UnsolicitedStatus services.
- Define additional objects and services that are unique to a particular class of device. The robot companion standard contains a definition of the VMD Stop service different from the core MMS Stop service.
- Define object naming and usage conventions for a particular class of device (for instance, standardized names and other naming conventions for common objects).

A companion standard, after proceeding through all the committees and work needed to become an ISO international standard, becomes a companion of the MMS standard as an additional part. The robot companion standard is ISO 9506 part 3, the NC companion standard will be ISO 9506 part 4, the PLC companion standard will be ISO 9506 part 5, and the PCS companion standard will be ISO 9506 part 6.

The reader should be aware that, for most systems, companion standards are not necessary even when using devices for which companion standards exist. The core MMS services defined in parts 1 and 2 of MMS provides sufficient standardization to achieve interoperability in most cases. Furthermore, aspects of the companion standards such as object naming, usage, and modeling conventions can be used in a core MMS application without having to implement the entire companion standard.

MMS Objects

MMS defines a variety of objects found in many typical devices and applications requiring real-time communications. Below is a list of these objects. For each object there are corresponding MMS services that let client applications access and manipulate those objects. The model for these objects and the available services will be described in more detail later.

- **VMD**. The device itself is an object
- **Domain**. Represents a resource (e.g., a program) within the VMD.
- **Program Invocation**. A runnable program consisting of one or more domains.
- **Variable**. An element of typed data (e.g., integer, floating point, array)
- **Type.** A description of the format of a variable's data.
- **Named Variable List**. A list of variables that is named as a list.
- **Semaphore**. An object used to control access to a shared resource.
- **Operator Station**. A display and keyboard for use by an operator.
- E**vent Condition**. An object that represents the state of an event.
- **Event Action**. Represents the action taken when an event condition changes state.
- **Event Enrollment**. Which network application to notify when an event condition changes state.
- **Journal**. A time based record of events and variables.
- **File.** A file in a filestore or fileserver.
- **Transaction**. Represents an individual MMS service request. Not a named object.

Object Attributes and Scope

Associated with each object is a set of attributes describing that object. MMS objects have a name attribute and other attributes that vary from object to object. Variables have attributes like, name, value, type, etc., while other objects, program invocations for instance, have attributes like name and current state.

Subordinate objects exist only within the scope of another object. For instance, all other objects are subordinate to, or contained within, the VMD itself. Some objects, such as the operator station object for example, may be subordinate only to the VMD. Some objects by be contained within other objects such as variables contained within a domain. This attribute of an object is called its *scope*. The object's scope also reflects the lifetime of an object. An object's scope may be defined to be:

- **VMD-Specific**. The object has meaning and exists across the entire VMD (is subordinate to the VMD). The object exists as long as the VMD exists.
- **Domain-Specific**. The object is defined to be subordinate to a particular domain. The object will exist only as long as the domain exists.
- **Application-Association-Specific** (also referred to as AA-Specific). The object is defined by the client over a specific application association and can only be used by that specific client. The object exists as long as the association between the client and server exists.

The name of a MMS object must also reflect the scope of the object. For instance, the object name for a domain-specific variable must not only specify the name of the variable within that domain but also the name of the domain. Names of a given scope must be unique. For instance, the name of a variable specific to a given domain must be unique for all domain specific variables in that domain. Some objects, such as variables, are capable of being defined with any of the scopes described above. Others for example, like semaphores, cannot be defined to be AA-specific. Still others, like operator stations for example, are only defined as VMD-specific. When an object such as a domain is deleted, all the objects subordinate to that domain must also be deleted.

The VMD Object

The VMD itself is also an object and has attributes associated with it. Some of the network visible attributes for a VMD are:

Capabilities

A capability of a VMD is a resource or capacity defined by the real device. There can be more than one capability to a VMD. A sequence of character strings represents these capabilities. They are defined by the implementor of the VMD and provide useful information about the real device or application.

Logical Status

Logical status refers to the status of the MMS communication system for the VMD. These can be: STATE-CHANGES-ALLOWED, NO-STATE-CHANGES-ALLOWED or ONLY-SUPPORT-SERVICES-ALLOWED.

Physical Status

Physical status refers to the status of all the capabilities taken as a whole. These can be equal to: OPERATIONAL, PARTIALLY-OPERATIONAL, INOPERABLE or NEEDS-COMMISSIONING.

VMD Support Services

See Volume 2 — Module 4

Status

The client uses this confirmed service to obtain the logical and physical status of the VMD. The Status and UnsolicitedStatus services also support access to implementation-specific status information (called *local detail*) defined by the implementor of the VMD.

UnsolicitedStatus

The server (VMD) uses this unconfirmed service to report its status to a client unsolicited by the client.

GetNameList

The client uses this confirmed service to obtain a list of named objects defined within the VMD.

Identify

The client uses this confirmed service to obtain information about the MMS implementation such as the vendor's name, model number, and revision level.

The VMD Execution Model

The VMD model has a flexible execution model that provides a definition of how the execution of programs by the MMS server can be controlled. Central to this execution model are the definitions of the domain and program invocation objects.

Domain Management

See Volume 2 — Module 6

The MMS domain is a named MMS object that is a representation of some resource within the real device. This resource can be anything appropriately represented as a contiguous block of untyped data (referred to as *load data*). In many typical applications, domains are used to represent areas of memory in a device. For instance, a PLC's ladder program memory is typically represented as a domain. Some applications allow blocks of variable data to be represented as both domains and variables. MMS provides no definition for, and imposes no constraints on, the content of a domain. To do so would be equivalent to defining a "real" object (i.e., the ladder program). The content of the domain is left to the implementor of the VMD. Besides name, some of the attributes associated with MMS domains are:

Capabilities

Each domain can optionally have a list of capabilities associated with it that conveys information about memory allocation, input/output characteristics and similar information about the real device. A sequence of implementor defined character strings represents the capabilities of a domain.

State

The state of a domain can be LOADING, COMPLETE, INCOMPLETE, READY, or IN-USE as well as other intermediate states.

Deletable

This attribute indicates whether the domain is deletable using the DeleteDomain service. A domain that can be downloaded is always deletable. Nondeletable domains are pre-existing and pre-defined by the VMD and cannot be downloaded.

Sharable

This attribute indicates if the domain can be shared by more than one program invocation (see the example batch controller on page 1-17).

MMS provides a set of services that allow domains to be uploaded from the device or downloaded to the device. The MMS domain services do not provide for partial uploads or downloads (except as potential error conditions) nor do they provide access to any subordinate objects within the domain. The set of Domain services is summarized below.

InitiateDownloadSequence, DownloadSegment, TerminateDownloadSequence

These services are used to download a domain. The InitiateDownloadSequence service commands the VMD to create a domain and prepare it to receive a download.

InitiateUploadSequence, UploadSegment, TerminateUploadSequence

These services are used to upload the contents of a domain to a MMS client.

DeleteDomain

A client uses this service to delete an existing domain, usually before initiating a download sequence.

GetDomainAttributes

This service is used to obtain the attributes of a domain.

RequestDomainDownload, RequestDomainUpload

A VMD uses these services to request that a client perform an upload or download of a domain in the VMD.

LoadDomainContent, StoreDomainContent

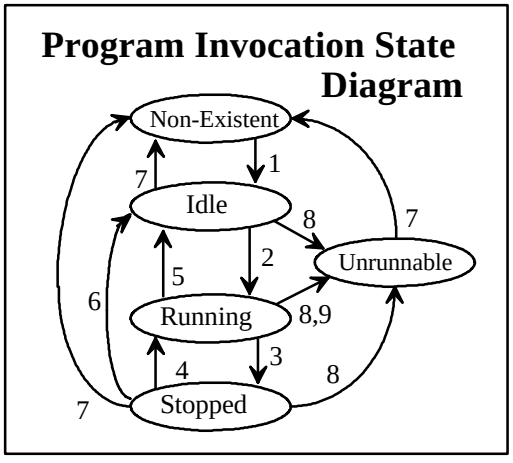
A VMD uses these services to tell a VMD to download (load) or upload (store) a domain from a file. The file may be local to the VMD or may be contained on an external file server.

Program Invocations

See Volume 2 — Module 6

It is through the manipulation of program invocations that a MMS client controls the execution of programs in a VMD. Program invocations can be started, stopped, reset, etc., by MMS clients. A program invocation is an execution thread that consists of a collection of one or more domains. Simple devices with simple execution structures may only support a single program invocation containing only one domain. More sophisticated devices and applications may support multiple program invocations containing several domains.

As an example, consider how the MMS execution model could be applied to a personal computer (PC). When the PC powers up, it downloads a domain called the operating system into memory. When you type the name of the program you want to run and hit the <return> key, the computer downloads another domain (the executable program) from a file and then creates and runs a program invocation consisting of the program and the operating system. The program by itself cannot be executed until it is bound to the operating system by the act of creating the program invocation.



MMS Clients use MMS services to cause state transitions in the program invocation as follows:

1. *CreateProgramInvocation service request.*
2. *Start service request.*
3. *Stop service request or program stop.*
4. *Resume service request.*
5. *End of program and Reusable=TRUE.*
6. *Reset service request.*
7. *DeleteProgramInvocation service request.*
8. *Kill service request or error condition.*

Figure 6: Program Invocation State Diagram

Besides the program invocation's name the attributes of a program invocation are:

State

The state of a program invocation can be NON-EXISTENT, IDLE, RUNNING, STOPPED, and UNRUNNABLE as well as other intermediate states.

List of Domains

The list of domains that comprise the program invocation.

Deletable

This indicates if the program invocation is deletable using the DeleteProgramInvocation service.

Reusable

Reusable program invocations automatically reenter the IDLE state when the program invocation arrives at the end of the program. Otherwise, program invocation in the RUNNING state must be Stopped, then Reset to bring it back to the IDLE state.

Monitored

Monitored program invocations use the MMS event management services to inform the MMS client when the program invocation leaves the RUNNING state. Monitored program invocations have an event condition object defined with the same name as the program invocation.

Execution Argument

This is a character string passed to the program invocation using the Start or Resume service. The execution argument passes data to the program invocation like parameters in a subroutine call.

The MMS services for program invocations allow clients to control the execution of VMD programs and to manage program invocations as follows:

CreateProgramInvocation

A client uses this to create a program invocation.

DeleteProgramInvocation

This service is used to delete a program invocation.

GetProgramInvocationAttributes

This service returns the attributes of the program invocation to the requesting client.

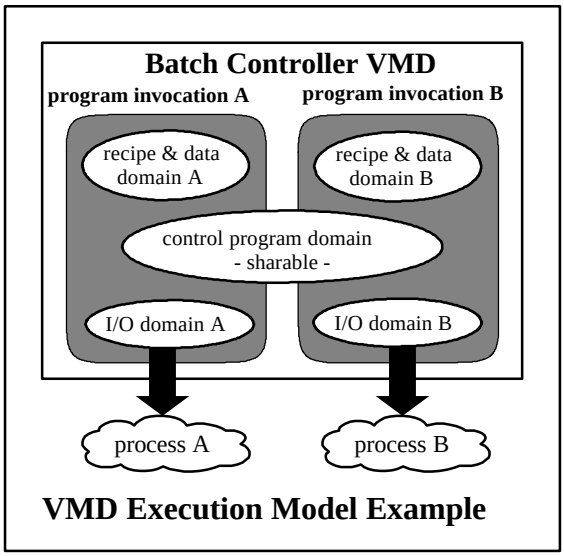
Start, Stop, Reset, Resume, Kill

A client uses these services to cause the program invocation to change states (see diagram).

Example Batch Controller

As an example of how the MMS execution model can be applied to a typical device, let us look at a VMD model for a simple batch controller. The diagram on the next page depicts how the VMD model could be applied to define a set of objects (such as domains, program invocations, variables) appropriate for a batch controller. This model will provide clients an appropriate method of controlling the batch process using MMS services. To startup and control these two processes, a MMS client using this controller would perform the following actions:

- Initiate and complete a domain download sequence for each domain: recipe and data domains A and B, I/O domains A and B, and the control program domain.
- Create program invocation A consisting of I/O domain A, the control program domain, and recipe and data domain A.
- Start program invocation A.
- Create program invocation B consisting of I/O domain B, the control program domain, and recipe and data domain B.
- Start program invocation B.



Our example batch controller allows us to control two identical batch oriented processes. An I/O domain, specific to each process ties the data in the corresponding recipe/data domain to the process. The control program domain is sharable and contains the control algorithm. The MMS client can create a separate program invocation to control each process. This allows the client to control the recipe and perform supervisory control (start, stop, etc.) of both processes independently form each other.

Figure 7: Batch Controller Example

The example above demonstrates the flexibility of the VMD execution model to accommodate a wide variety of real world situations. Additional examples might be a loop controller where each loop is represented by a single domain and where the control loop algorithm (e.g., PID) is represented by a sharable separate domain. Process variables, setpoints, alarm thresholds, etc., could be represented by domain (control loop) specific variables. A program invocation would consist of the control loop domains and their algorithm domains needed to control the process.

The Variable Access Model

See Volume 2 — Module 5

MMS Variables

A real variable is an element of *typed* data contained within a VMD. A MMS variable is a virtual object that represents a mechanism for MMS clients to access the real variable. The distinction between the real variable (containing the value) and the virtual variable (representing the access path to the variable) is important. When a MMS variable is deleted, only the access to the variable is deleted, not the actual real variable. When a MMS variable is created, it is the access path that is created, not the real variable. MMS defines two types of virtual objects for describing variable access:

Unnamed Variable Object

An unnamed variable object describes the access to the real variable by using a device specific *address*. MMS includes unnamed variable objects primarily for compatibility with older devices that are not capable of supporting names. An unnamed variable object is a direct mapping to the underlying real variable located at the specified address. The attributes of the unnamed variable object are:

Address

This is the key attribute of the unnamed variable object.

MMS Deletable

This attribute is always FALSE for an unnamed variable object. Unnamed variable objects cannot be deleted because they are a direct representation of the “real” variable located at a specific address in the VMD.

Type Description

This attribute describes the type (format and range of (values)) of the variable.

Named Variable Object

The named variable object describes the access to the real variable by using a MMS object name. MMS clients need only know the name of the object in order to access it. Remember that the name of a MMS variable also specifies the scope (see page 1-14) of the object. Besides the name, a named variable object has the following attributes:

MMS Deletable

This attribute indicates if access to the variable can be deleted using the DeleteVariableAccess service.

Type Description

This attribute describes the type (format and range of values) of the variable.

Access Method

If the access method is PUBLIC, it means that the underlying address of the named variable object is visible to the MMS Client. In this case the same variable can be accessed as an unnamed variable object.

Addresses

A MMS variable address can take on one of several forms. The specific form used by a specific VMD, and the specific conventions used by that address form, are determined by the implementor of the VMD based upon what is most appropriate for that device. The possible forms for a MMS variable address are:

- **Numeric**. A numeric address is represented by an unsigned integer number (e.g., 103).
- **Symbolic**. A symbolic address is represented by a character string (e.g., “R001” or “N7:0”).
- **Unconstrained**. An unconstrained address is represented by a untyped string of bytes.

In general, it is recommended that applications use named variable objects instead of the addresses of unnamed variable objects wherever feasible. Address formats vary widely from device to device. Furthermore, in some computing environments, the addresses of variables can change from one run-time to the next. Names provide a more intuitive, descriptive and device independent access method than addresses.

Named Variable Lists

MMS also defines a *named variable list* object that provides an access mechanism for grouping both named and unnamed variable objects into a single object for easier access. A named variable list is accessed by a MMS client by specifying the name (which also specifies its scope) of the named variable list.

When the VMD receives the Read service request from a client, it reads all the individual objects in the list and returns their value within the individual elements of the named variable list. Because the named variable list object contains independent subordinate objects, a positive confirmation to a Read request for a named variable list may indicate only partial success. The success or failure status of each element in the confirmation must be examined by the client to ensure that all the underlying variable objects were accessed without error. Besides its name and the list of underlying named and unnamed variable objects, named variable list objects also have a MMS deletable attribute that indicates whether the named variable list can be deleted using the DeleteNamedVariableList service request.

Named Type Object

The type of a variable indicates its format and the possible range of values that the variable can take. Examples of type descriptions include 16-bit signed integer, double precision floating point, 16 character string, etc. MMS allows the type of a variable to be either 1) *described* or 2) defined as a separate named object called a *named type*. A described type is not an object. It is a binary description of the type in a MMS service request (Read, Write, or InformationReport) that uses the Access by Description service option. The named type object allows the types of variables to be defined and managed separately. This can be particularly useful for systems that also use the DefineNamedVariable service to define names and types for unnamed variable objects.

Other attributes of the named type object include:

MMS Deletable

This parameter indicates if the named type can be deleted using a DeleteNamedType service request.

Type Description

This describes the type in terms of complexity (simple, array, or structured), format (such as integer, floating point), and the size or range of values (16-bit, 8 characters, etc.).

Type Description

The specification for a MMS type description is very flexible and can describe virtually any data format in use today. As mentioned earlier, the type description specifies the format of a variable and the range of values that the variable can represent. MMS defines three basic formats for types: 1) *simple*, 2) *array* and 3) *structured*:

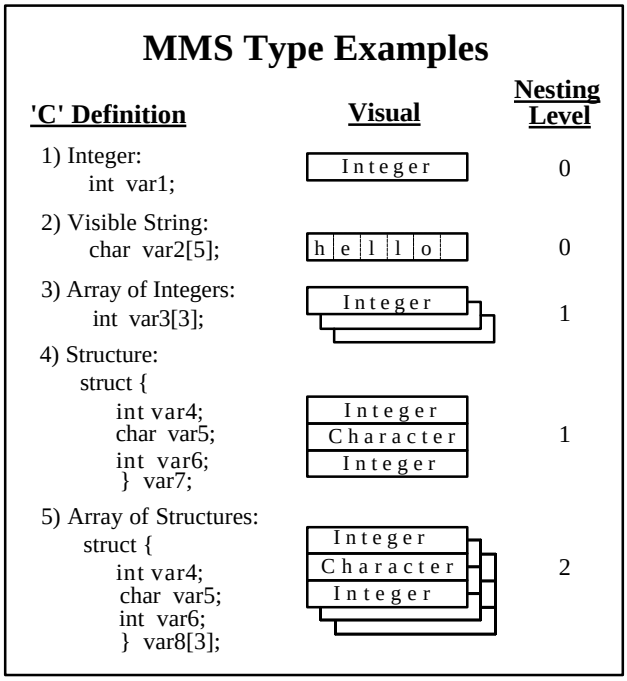


Figure 8: MMS Type Examples

- **Simple.** Simple types are the most basic types and cannot be broken down into a smaller unit using MMS. The other type forms (arrays and structures) are constructed types that can eventually be broken down into simple types. Simple type descriptions generally consist of the *class* and *size* of the type. The size parameter is usually defined in terms of the number of bits or bytes that a variable of that type would comprise in memory. The various classes of simple types defined by MMS consists of:

  BOOLEAN. Booleans are generally represented as a single byte with either a zero (**SD_FALSE**) or non-zero (**SD_TRUE**) value. There is no size parameter for Boolean types.

  BIT STRING. A Bit String is a sequence of bits. The size of a Bit String indicates the number of bits in the Bit String.

BOOLEAN ARRAY. A Boolean Array is also a sequence of bits where each bit represents TRUE (1) or FALSE (0). This differs from an array of Booleans in that each element in a Boolean Array is represented by a single bit while each element in an array of Booleans is represented by a single byte. The size parameter specifies the number of Booleans (number of bits) in the Boolean Array.

INTEGER. MMS Integers are signed integers. The size parameter specifies the number of bits of the integer.

UNSIGNED. The Unsigned type is identical to the Integer type except that it is not allowed to take on a negative value. Because the most significant bit of an Integer is essentially a sign bit, an Unsigned with a size of 16 bits can only represent 15-bits of values or values of 0 through 32,767.

FLOATING POINT. The MMS definition for floating point is modeled after the IEEE 754 standard but can accommodate any number of bits for the format and exponent width including the single and double precision floating point formats commonly in use today.

REAL. This type is a representation of floating point numbers conforming to the ISO 8824 standard.

OCTET STRING. An Octet String is a sequence of bytes (called an “octet” in ISO terminology) with no constraint on the value of the individual bytes. The size of an Octet String is the number of bytes in the string.

VISIBLE STRING. Unlike the Octet String type, the Visible String type only allows each byte to contain a printable character. Although these character sets are defined by ISO 10646, they are compatible with the common ASCII character set. The size of the Visible String is the number of bytes in the string.

GENERALIZED TIME. This is a representation of time specified by ISO 8824. It provides millisecond resolution of date and time.

BINARY TIME. This is a time format specified by MMS. It represents either 1) time of day by a value that is equal to the number of milliseconds from midnight or 2) date and time by a value that is equal to the time of day and the number of days since January 1, 1984.

BCD. Binary Coded Decimal format is where four-bits are used to hold a binary value of a single digit of zero to ten. The size parameter is the number of decimal digits that the BCD value can represent.

OBJECT IDENTIFIER. This is a special class of object defined by ISO 8824 that is used to define network objects.

- **Array.** An array type defines a variable that consists of a sequence of multiple identical (in format not value) elements. Each element in an array can also be an array or even a structured or simple variable. MMS allows for arbitrarily complex nesting of arrays and structures. The level of complexity that a VMD can support is defined by its *nesting level*. An array of simple variables (e.g., an array of Integers) requires a nesting level of 1. An array of an array or an array of structured variables, each consisting of simple variables, requires a nesting level of 2.
- **Structures.** A structured type defines a variable that consists of a sequence of multiple, but not necessarily identical, elements. Each individual element in a structure can be of a simple type, an array, or another structure. MMS allows for arbitrarily complex nesting of structures and arrays. A structured variable consisting of individual simple elements requires a nesting level of 1. A structured variable consisting of one or more arrays of structures containing simple variables requires a nesting level of 3.

Variable Access Services

Read

MMS clients use this service to obtain the values of named, unnamed, or named variable list objects (hereinafter “Variables”).

Write

MMS clients use this service to change the values of Variables.

InformationReport

A VMD uses this service to report the values of Variables to a MMS client in an unsolicited manner. It is roughly equivalent to sending a Read response to a client without the client having issued the Read request. The InformationReport service can be used to eliminate polling by client applications or as an alarm notification mechanism. The VMD could directly report changes in the value of Variables, alarm conditions, or even changes in state of the VMD or program invocations to clients by using the InformationReport service. MMS does not require the use of the InformationReport service for this purpose; it is simply one of the many potential applications of the service.

GetVariableAccessAttributes

MMS clients use this service to obtain the access attributes of a single named or unnamed variable object. The access attributes are the type (either a named type or type description), the MMS deletable attribute and the address for unnamed variables. The address of named variables is optional and is only returned if the address is known and visible.

DefineNamedVariable

MMS clients use this service to create a named variable object corresponding to a real VMD variable that can be represented by an unnamed variable object. Once defined, subsequent access to the unnamed variable object can be made using the named variable. This service also allows the MMS client to optionally specify a type definition for the named object that may be different from the type inherently defined for the unnamed variable object.

DeleteVariableAccess

This service is used to delete named variable objects where the MMS deletable attribute is TRUE. The service provides options for deleting only specific named variable objects or all named variable objects of a specified scope (VMD, domain, or AA-specific).

DefineNamedVariableList, GetNamedVariableListAttributes, DeleteNamedVariableList

These services are used to create, delete and obtain the attributes (i.e., the list of underlying named and unnamed variable objects) of named variable list objects.

DefineNamedType

MMS clients use this service to create a new named type by specifying the type name (including scope), MMS deletable and type description attributes.

DeleteNamedType

This service is used to delete an existing named type where the MMS deletable attribute is TRUE. The service provides options for deleting only specific named type objects or all named type objects of a specified scope (VMD, domain, or AA-specific).

GetNamedTypeAttributes

MMS clients use this service to determine the MMS deletable and type description attributes of a specific named type object.

Variable Access Service Features

The Read, Write and InformationReport services provide several features for accessing Variables. The use of these service features, described below, by MMS clients can provide enhanced performance and very flexible access to MMS Variables.

- **Access by Description** is supported for unnamed variable objects only. It allows the MMS client to describe the variable by specifying both an address and a type description. These variables are called *described variables*. The described type may be different from the type inherently defined for the unnamed

variable object. This can be useful for accessing data in devices where the device's memory organization is simplistic. For example, many PLCs represent their data memory as a large block of 16-bit registers (essentially signed integers). Some applications may store an ASCII string data in these registers. By using a described variable, a MMS client can have the data stored in these registers returned to it as a string instead of as a block of signed integers.

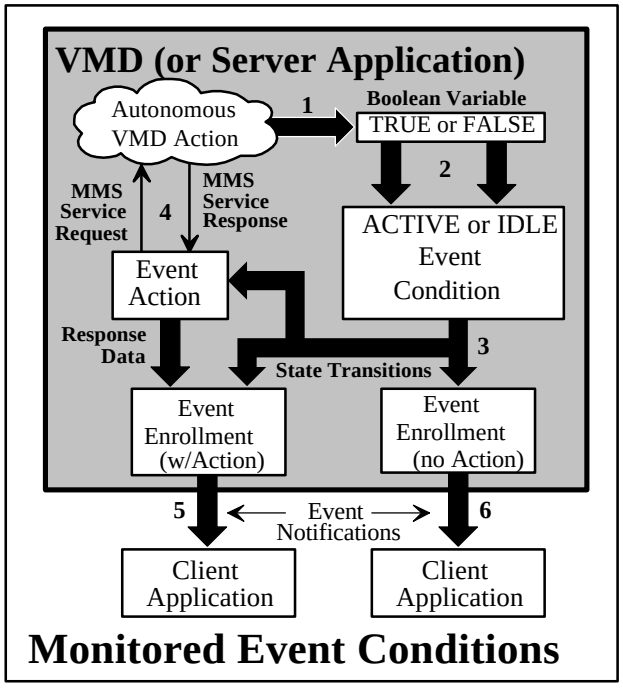
- **List of Variables.** This feature allows a list of named variable, unnamed variable, and named variable list objects to be accessed in a single MMS Read, Write, or InformationReport service. Care must be taken by the client to ensure that the resultant MMS service request message does not exceed the maximum message size (or *maximum segment size*) supported by the VMD. This option also requires that a client examine the entire response for the success or failure for each individual element in the list of Variables.
- **Access Specification in Result** is an option for the Read service that allows a MMS client to request that the variable's access specification be returned in the Read response. The access specification would consist of the same information that would be returned by a GetVariableAccessAttributes service request.
- **Alternate Access** allows a MMS client to 1) partially access only specified elements contained in a larger arrayed and/or structured variable, and 2) rearrange the ordering of the elements contained in structured variables.

The Event Management Model

See Volume 3 — Module 7

In a real sense an event, or an alarm, is easy to define. Most people have an intuitive feel for what can comprise an event within their own area of expertise. For instance, in a process control application it is common for a control system to generate an alarm when the process variable (e.g., temperature) exceeds a certain preset limit called the high alarm threshold. In a power distribution application an alarm might be generated when the difference in the phase angle of the current and voltage waveforms of a power line exceeds a certain number of degrees.

The MMS event management model provides a framework for accessing and managing the network communication aspects of these kinds of events. This is accomplished by defining three named objects. These represent 1) the state of an event (*event condition*), 2) who to notify about the occurrence of an event (*event enrollment*) and 3) the action that the VMD should take upon the occurrence of an event (*event action*).



A Monitored event condition has a Boolean variable associated with it that the VMD sets (1) via some form of the local autonomous action. The VMD periodically evaluates this variable (2) to determine the state of the event condition. MMS Clients "enroll"" to be notified of event condition state transitions (3) by defining an event enrollment. If an event action is defined for the event enrollment, the VMD obtains the response (4) to the event action (a MMS service request) and inserts the response data in the event notification (5) that is sent to the client. Event enrollments without an event action have their event notifications sent to the client (6) without any event action response data.

Figure 9: Monitored Event Conditions

For many applications, the communication of alarms can be implemented by using MMS services other than the event management services. For instance, a simple system can notify a MMS client about the fact that a process variable has exceeded some preset limit by sending the process variable's value to a MMS client using the InformationReport service. Other schemes using other MMS services are also possible. When the application is more complex and requires a more rigorous definition of the event environment to ensure interoperability, the MMS event management model should be used.

Event Condition Object

A MMS event condition object is a named object that represents the current state of some real condition within the VMD. It is important to note that MMS does not define the VMD action (or programming) that causes a change in state of the event condition. In the process control example given above, an event condition might reflect an IDLE state for when the process variable was not exceeding the value of the high alarm threshold and an ACTIVE state when the process variable did exceed the limit. MMS does not explicitly define the mapping between the high alarm limit and the state of the event condition. Even if the high alarm limit is represented by a MMS variable, MMS does not define the necessary configuration or programming needed to create the mapping between the high alarm limit and the state of the event condition. From the MMS point of view, the change in state of the event condition is caused by some autonomous action on the part of the VMD not defined by MMS.

The MMS event management model defines two classes of event conditions:

- **Network Triggered**. A network triggered event condition is triggered when a MMS client specifically triggers it using the TriggerEvent service request. Network triggered events do not have a state (their state is always DISABLED). They are useful for allowing a MMS client to control the execution of event actions and the notifications of event enrollments.
- **Monitored**. A monitored event condition has a state attribute that the VMD sets based upon some local autonomous action. Monitored event conditions can have a Boolean variable associated with them that is used by the VMD to evaluate the state. The VMD periodically evaluates this variable. If the variable is evaluated as TRUE, the VMD sets the event condition state to ACTIVE. When the Boolean variable is evaluated as FALSE, the VMD sets the event condition state to IDLE. Event conditions that are created as a result of a CreateProgramInvocation request with the Monitored attribute TRUE, are monitored event conditions but they do not have an associated Boolean variable.

Besides the name of the event condition (an object name that also reflects its scope) and its class (Network Triggered or Monitored), MMS defines the following attributes for both network triggered and monitored event conditions:

MMS Deletable

This attribute indicates if the event condition can be deleted by using a DeleteEventCondition service request.

State

This attribute reflects the state of the event condition and can be IDLE, ACTIVE, or DISABLED. Network triggered events are always DISABLED.

Priority

This attribute reflects the relative priority of an event condition object with respect to all other defined event condition objects. Priority is a relative measure of the VMD's processing priority when it comes to evaluating the state of the event condition as well as the processing of event notification procedures that are invoked when the event condition changes state. A value of zero (0) is the highest priority. A value of 127 is the lowest priority. A value of 64 is the "normal" priority.

Severity

This attribute reflects the relative severity of an event condition object with respect to all other defined event condition objects. Severity is a relative measure of the effect that a change in state of the event condition can have on the VMD. A value of zero (0) is the highest severity. A value of 127 is the lowest. A value of 64 is for event conditions with “normal” severity.

Additionally, MMS also defines the following attributes for Monitored event conditions only:

Monitored Variable

This is a reference to the underlying Boolean variable whose value the VMD evaluates in determining the state of an event condition. It can be either a named or unnamed variable object. If it is a named object it must be a variable with the same name (and scope) of the event condition. If the event condition object is locally defined or it was defined using a CreateProgramInvocation request with the monitored attribute TRUE, then the value of the monitored variable reference would be equal to UNSPECIFIED. If the monitored variable is deleted, then the value of this reference would be UNDEFINED and the VMD should disable its event notification procedures for this event condition.

Enabled

This attribute reflects whether a change in the value of the monitored variable (or the state of the associated program invocation if applicable) should cause the VMD to process the event notification procedures for the event condition (TRUE) or not (FALSE). A client can disable an event condition by changing this attribute using an AlterEventConditionMonitoring service request.

Alarm Summary Reports

This attribute indicates whether (TRUE) or not (FALSE) the event condition should be included in alarm summary reports in response to a GetAlarmSummaryReport service request.

Evaluation Interval

This attribute specifies the maximum amount of time, in milliseconds, between successive evaluations of the event condition by the VMD. The VMD may optionally allow clients to change the evaluation interval.

Time of Last Transition to Active, Time of Last Transition to Idle

These attributes contain either the time of day or a time sequence number of the last state transitions of the event condition. If the event condition has never been in the IDLE or ACTIVE state, then the value of the corresponding attribute shall be UNDEFINED.

Event Condition Services

DefineEventCondition, DeleteEventCondition, GetEventConditionStatus

MMS clients use this service to create event condition objects, to delete event condition objects (if the MMS Deletable attribute is TRUE) and to obtain the static attributes of an existing event condition object respectively.

ReportEventConditionStatus

A MMS client uses this service to obtain the dynamic status of the event condition object. This includes its state, the number of event enrollments enrolled in the event condition, whether it is enabled or disabled, and the time of the last transitions to the active and idle states.

AlterEventConditionMonitoring

A MMS client uses this service to alter the priority, enable or disable the event condition, enable or disable alarm summary reports and to change the evaluation interval if the VMD allows the evaluation interval to be changed.

GetAlarmSummary

A MMS client uses this service to obtain event condition status and attribute information about groups of event conditions. The client can specify several filters for determining which event conditions to include in an alarm summary.

Event Actions

An event action is a named MMS object that represents the action that the VMD will take when the state of an event condition changes. An event action is optional. When omitted, the VMD would execute its event notification procedures without processing an event action. An event action, when used, is always defined as a confirmed MMS service request. The event action is attached or linked with an event condition when an event enrollment is defined. For example, an event action might be a MMS Read request. If this event action is attached to an event condition (by being referenced in an event enrollment), when the event condition changes state and the event condition is enabled, the VMD would execute this Read service request just as if it had been received from a client. Except that the Read response (either positive or negative) is included in the EventNotification service request that is sent to the MMS client defined for the event enrollment. A confirmed service request must be used such as Start, Stop, Read. Unconfirmed services such as InformationReport, UnsolicitedStatus, and EventNotification, and other services that must be used in conjunction with other services (e.g., domain upload-download sequences) cannot be used as event actions. Besides its name, an event action has the following attributes:

MMS Deletable

When this attributes is TRUE, it indicates that the event action can be deleted using a DeleteEventAction service request.

Service Request

This attribute is the MMS confirmed service request that the VMD will process when the event condition that the event action is linked with changes state.

Event Action Services

DefineEventAction, DeleteEventAction, GetEventActionAttributes

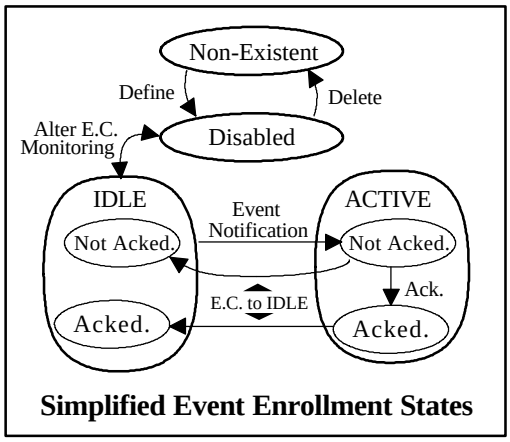
MMS clients use this service to create, delete (if the MMS Deletable attribute is TRUE) and to obtain the attributes of an event action object respectively.

ReportEventActionStatus

A MMS client uses this service to obtain a list of names of event enrollments that have referenced a given event action.

Event Enrollments

The event enrollment is a named MMS object that ties all the elements of the MMS event management model together. The event enrollment represents a request on the part of a MMS client to be notified about changes in state of an event condition.



Event enrollments have major states (IDLE, ACTIVE) that reflect the event condition (E.C.). Minor states reflect the status of the event acknowledgment process.

Figure 10: Simplified Event Enrollment States

When an event enrollment is defined, references are made to an event condition, an event action (optionally) and the MMS client to which EventNotification should be sent. Besides its name, the attributes of an event enrollment are:

MMS Deletable

If this attribute is TRUE, it indicates that the event enrollment can be deleted using a DeleteEventEnrollment service request.

Event Condition

This attribute is the name of the event condition about which the event enrollment will be notified of changes in state.

Transitions

This attribute indicates the state transitions of the event condition for which the VMD should execute its event notification procedures as follows:

- DISABLED-TO-ACTIVE
- DISABLED-TO-IDLE
- IDLE-TO-ACTIVE
- IDLE-TO-DISABLED

Notification Lost

If this attribute is TRUE, it means that the VMD could not successfully complete its event notification procedures due either to 1) some local resource constraint or problem or 2) because the VMD could not establish an association to the client specified in the event enrollment definition. Further transitions of the event condition will be ignored for this event enrollment as long as these problems persist.

Event Action

This optional attribute is a reference to the event action that should be processed by the VMD for those state transitions of the event condition specified by the event enrollment's transitions attribute.

Client Application

This attribute is a reference to the MMS client to which the EventNotification service requests should be sent for those transitions of the event condition specified by the event enrollment's transitions attribute. This attribute should only be defined if the VMD supports third party services. This attribute is omitted if the duration of the event enrollment is CURRENT.

Duration

This attribute indicates the lifetime of the event enrollment. A duration of CURRENT means the event enrollment is only defined for the life of the association between the MMS client and the VMD (similar to an object with application association specific scope). If the association between the VMD and the client is lost and there is no client application reference attribute for the event enrollment (client application reference is omitted for event enrollments with duration = CURRENT), then the VMD is not capable of re-establishing an application association in order to send an EventNotification to the client. If the duration of the event enrollment is PERMANENT, then the application association between the VMD and the client can be terminated without affecting the event enrollment. In this case, when a specified state transition occurs, the VMD will automatically establish an application association with the specified client.

State

The state of an event enrollment indicates IDLE, ACTIVE, DISABLED and a variety of other states that represent the status of the event notification procedures being executed by the VMD.

Alarm Acknowledgement Rules

This attribute specifies the rules of alarm acknowledgment that the VMD should enforce when determining the state of the event enrollment. If an acknowledgment to an EventNotification service request is required, the act of acknowledging (or not acknowledging) the Event Notification shall affect the state of the event enrollment. The various alarm acknowledgment rules are summarized as follows:

- **NONE**. No acknowledgments are required and they shall not affect the state of an event enrollment. These types of event enrollments are not included in alarm enrollment summaries.
- **SIMPLE**. Acknowledgment shall not be required but acknowledgments of transitions to the ACTIVE state shall affect the state of the event enrollment.
- **ACK-ACTIVE**. Acknowledgment of event condition transitions to the ACTIVE state shall be required and shall affect the state of the event enrollment. Acknowledgments of other transitions are optional and shall not affect the state of the event enrollment.
- **ACK-ALL**. Acknowledgments are required for all transitions of the event condition to the ACTIVE or IDLE state and shall affect the state of the event enrollment.

Time Active Acked, Time Idle Acked

These attributes reflect the time of the last acknowledgment of the Event Notification for state transitions in the event condition to the ACTIVE or IDLE state corresponding to the event enrollment.

Event Enrollment Services

DefineEventEnrollment, DeleteEventEnrollment, GetEventEnrollmentAttributes

MMS clients use these services to create, delete (if the MMS Deletable attribute is TRUE) and to obtain the static attributes of an event enrollment object.

ReportEventEnrollmentStatus

This service allows the client to obtain the dynamic attributes of an event enrollment including the notification lost, duration, alarm acknowledgment rule and state attributes.

AlterEventEnrollment

This service allows the client to alter the transitions and alarm acknowledgment rule attributes of an event enrollment.

GetAlarmEnrollmentSummary

This service allows a MMS client to obtain event enrollment and event condition information about groups of event enrollments. The client can specify several filters for determining which event enrollments to include in an alarm enrollment summary.

Event Notification Services

MMS provides services for notifying clients of event condition transitions and acknowledging those event notifications as follows:

EventNotification

This is an unconfirmed service that is issued by the VMD to the MMS client to notify the client about event condition transitions that were specified in an event enrollment. There is no response from the client. The acknowledgment of the notification is handled separately. The EventNotification service would include a MMS confirmed service response (positive or negative) if an event action was defined for the event enrollment.

AcknowledgeEventNotification

A MMS client uses this confirmed service to acknowledge an EventNotification sent to it by the VMD. The client specifies the event enrollment name, the acknowledgment state, and the transition time parameters that were in the EventNotification request that is being acknowledged.

TriggerEvent

This service is used to trigger a Network Triggered event condition. It gives the client a mechanism by which it can invoke event action and event notification processing by the VMD. For instance, a client can define an event condition, event action, and event enrollments that refer to other MMS clients. When the defining client issues a TriggerEvent service request to the VMD it will cause the VMD to execute a MMS service request (the event action) and send these results to other MMS clients using the EventNotification service.

The Semaphore Management Model

See Volume 3 — Module 7

In many real-time systems there is a need for a mechanism by which an application can control access to a system resource. An example might be a workspace that is physically accessible to several robots. Some means to control which robot (or robots) can access the workspace is needed. MMS defines two types of *semaphores* for these types of applications: 1) *Token Semaphores* and 2) *Pool Semaphores*.

Token Semaphores

A token semaphore is a named MMS object that can be a representation of some resource within the control of the VMD to which access must be controlled. A token semaphore is modeled as a collection of tokens that MMS clients take and relinquish control of using MMS services. This allows both multiple or exclusive ownership of the semaphore. When a MMS client owns the token, it provides some level of access to the underlying resource. An example might be where two users want to change a setpoint for the same control loop at the same time. These users could use a MMS token semaphore containing only one token to represent the control loop in order to coordinate their access to the setpoint. When the user “owns” the token, they can change the setpoint. The other would have to wait until ownership is relinquished.

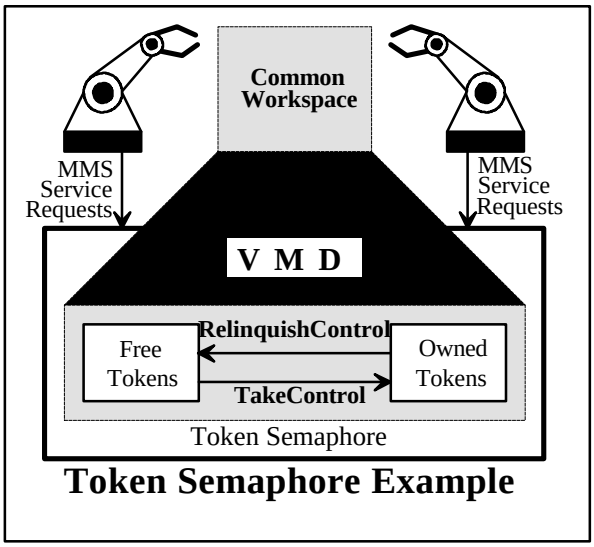


Figure 11: Token Semaphore Example

A token semaphore can also be used for the sole purpose of coordinating the activities of two MMS clients without representing any real resource. This kind of “virtual” token semaphore looks and behaves the same except that they can be created and deleted by MMS clients using the DefineSemaphore service.

Because semaphores either represent a real resource or are used for the purpose of coordinating activities between two or more MMS clients, the scope of a semaphore cannot be AA-Specific. If an object’s scope is AA-Specific there can be only one client. Also, AA-Specific objects only exist as long as the application association exists while real resources must exist outside the scope of any given application association.

In addition to its name, the token semaphore has the following attributes:

Deletable

If TRUE, it means that the semaphore does not represent any real resource within the VMD and can therefore be deleted by a MMS client using the DeleteSemaphore service.

Number of Tokens

This attribute indicates the total number of tokens contained in the token semaphore.

Owned Tokens

This attribute indicates the number of tokens whose associated semaphore entry state is OWNED.

Hung Tokens

This attribute indicates the number of tokens whose associated semaphore entry state is HUNG.

Pool Semaphores

A pool semaphore is similar to a token semaphore except that the individual tokens are identifiable and have a name associated with them. These *named tokens* can optionally be specified by the MMS client when issuing TakeControl requests. The pool semaphore itself is a MMS object. The named tokens contained in the pool semaphore are not MMS objects. They are representations of a real resource in much the same way an unnamed variable object is. The name of the pool semaphore is separate from the names of the named tokens. Pool semaphores can only be used to represent some real resource within the VMD. Therefore, pool semaphores cannot be created or deleted using MMS service requests and cannot be AA-Specific in scope.

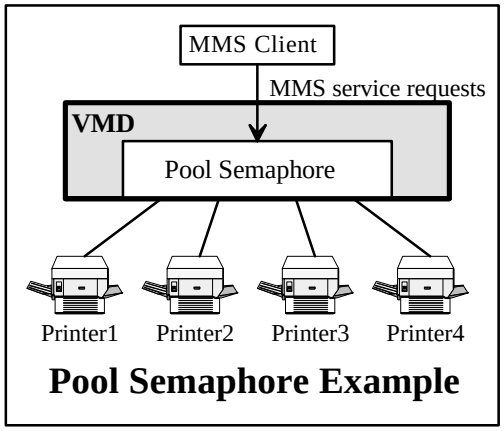


Figure 12: Pool Semaphore Example

In addition to the name of a pool semaphore, the following attributes also are defined by MMS:

Free Named Tokens

These are the named tokens for which no associated semaphore entry exists.

Owned Named Tokens

These are the named tokens for which the associated semaphore entry state is OWNED.

Hung Named Tokens

These are the named tokens for which the associated semaphore entry is HUNG.

Semaphore Entry

When a MMS client issues a TakeControl request for a given semaphore the VMD creates an entry in an internal queue that is maintained for each semaphore. Each entry in this queue is called a *semaphore entry*. The attributes of a semaphore entry are visible to MMS clients and provide information about the internal semaphore processing queue in the VMD. The semaphore entry is not a MMS object. It only exists from the receipt of the TakeControl indication by the VMD until the token control of the semaphore is relinquished or if the VMD responds negatively to the TakeControl request. Several of the attributes of the semaphore entry are specified by the client in the TakeControl request. These attributes are:

Entry ID

This is a number assigned by the VMD to distinguish one semaphore entry from another. The Entry ID is unique for a given semaphore.

Named Token

This is valid only for pool semaphores. It contains the named token that was optionally requested by the client in a TakeControl request or, if the semaphore entry is in the OWNED or HUNG state, it is the named token that the VMD assigned as a result of a TakeControl request.

Application Reference

This is a reference to the MMS client application that issued the TakeControl request that created the semaphore entry.

Priority

This attribute indicates the priority of the semaphore entry with respect to other semaphore entries. Priority is used to decide which semaphore entry in the QUEUED state will be granted a token (or named token) when multiple requests are outstanding. The value (0 = highest priority, 64 = normal priority, and 127 = lowest priority) is specified by the client in the TakeControl request.

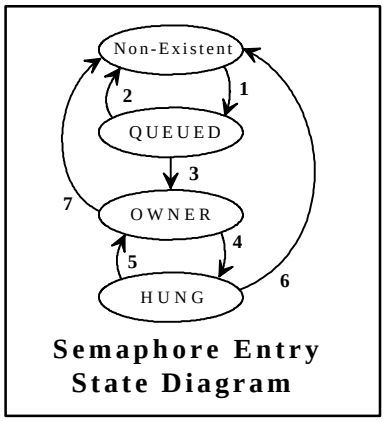
Entry State

The entry state attribute represents the relationship between the MMS client and the semaphore by one of the following values:

QUEUED. This means that a TakeControl request has been received but has not been responded to by the VMD. The client is waiting for control of the semaphore.

OWNED. The VMD has responded positively to the TakeControl request and the client now owns the token (or named token).

HUNG. This state means that the application association over which the MMS client issued the TakeControl request has been lost and the Relinquish if Connection Lost attribute is FALSE. A MMS client can take control of a semaphore entry in the HUNG state by issuing a TakeControl request with the *preempt* option TRUE and by specifying the MMS client to preempt (using the application reference attribute).



A semaphore entry is created each time a client attempts to take control of a semaphore. The semaphore entry reflects the state of the relationship between the client and the semaphore. State transitions can be caused by local autonomous action by the VMD or the following events:

1. *TakeControl request received.*
2. *Timeout, Cancel request or abort.*
3. *Semaphore available (token free).*
4. *Application association aborted and Rel.if.Conn.Lost = FALSE.*
5. *TakeControl with preempt.*
6. *Control timeout.*
7. *a) RelinquishControl request received, or b) control timeout, or c) application association aborted and Rel.if.Conn.Lost = TRUE.*

Figure 13: Semaphore Entry State Diagram

Relinquish if Lost

If this attribute is TRUE the VMD will relinquish control of the semaphore if the application association for the MMS client that owned the token for this semaphore entry is lost or aborted. If FALSE, the semaphore entry will enter the HUNG state if the application association is lost or aborted.

Control Timeout

This attribute is specified by the client in the TakeControl request and indicates how many milliseconds the client should be allowed to control the semaphore once control is granted. If the client has not relinquished control using a RelinquishControl request when the control timeout expires, the semaphore entry will be deleted and control of the semaphore be relinquished. If the control timeout attribute is omitted in the TakeControl request then no control timeout will apply for that semaphore entry.

Abort on Timeout

This attribute is specified by the client in the TakeControl request and indicates if the VMD should abort the application association with the owner client upon a control timeout.

Acceptable Delay

This attribute is specified by the client in the TakeControl request and indicates how many milliseconds the client is willing to wait for control of the semaphore. If control is not granted during this time, the VMD will respond negatively to the TakeControl request. If the acceptable delay attribute is omitted from the TakeControl request then the client is willing to wait indefinitely.

Semaphore Services

TakeControl

A MMS client uses this service to request control of a semaphore.

RelinquishControl

A MMS client uses this service to release control over a semaphore that the client currently has control of.

DefineSemaphore, DeleteSemaphore

MMS clients use these services to define and delete token semaphores that are used solely for coordinating the activities of two or more clients.

ReportSemaphoreStatus, ReportPoolSemaphoreStatus

These services are used to obtain the status (number of total, owned and hung tokens) of token and pool semaphores.

ReportSemaphoreEntryStatus

A MMS client uses this service to obtain the attributes of semaphore entries corresponding to a specified state.

Other MMS Objects

Operator Stations

An *operator station* is a MMS object that can be used to represent character based input and output devices that may be attached to the VMD to communicate with an operator local to the VMD. MMS defines three types of operator stations:

- ***Entry***. An entry only operator station consists of an input device only. This may be a keyboard or perhaps a bar code reader. The input data must be of the type Visible String consisting of alpha-numeric characters only.
- ***Display***. A display only operator station consists of a character based output display that can display Visible String data (no graphics or control characters).
- ***Entry-Display***. This type of operator station consists of both an entry station and a display station.

Because the operator station is a representation of a physical feature of the VMD, it exists beyond the scope of any domain or application association. Therefore, MMS clients access the operator station by name without scope. MMS allows for any number of operator stations for a given VMD. The services used by MMS clients to perform operator communications are:

Input

MMS clients use this service to obtain a single input string from an input device. The service has an option for displaying a sequence of prompts on the display if the operator station is an entry-display type of operator station.

Output

This service is used to display a sequence of output strings on the display of the operator station.

Journals

A MMS *journal* represents a log file that contains a collection of records (called *journal entries*) that are organized by time stamps. Journals are used to store time based records of tagged variable data, user generated comments or combination of events and tagged variable data. Journal entries contain a time stamp that indicates when the data in the entry was produced (not when the journal entry was made). This allows MMS journals to be used for applications where a sample of a manufactured product is taken at one time, analyzed in a laboratory off-line, and then later placed into the journal. In this case the journal entry time stamp would indicate when the sample was taken.

MMS clients read the journal entries by specifying the name of the journal (which can be VMD-Specific or AA-Specific only) and either 1) the date/time range of entries that the client wishes to read or 2) by referring to the entry ID of a particular entry. The entry ID is a unique binary identifier assigned by the VMD to the journal entry when it is placed into the journal. Each entry in a journal can be one of the following types:

Annotation

This type of entry contains a textual comment. This is typically used to enter a comment regarding some event or condition that had occurred in the system.

Data

This type of entry would contain a list of variable tags and the data associated with those tags at the time indicated by the time stamp. Each variable tag is a 32-character name that does not necessarily refer to a MMS variable (although it might).

Event-Data

This type of entry contains both variable tag data and event data. Each entry of this type would include the same list of variable tags and associated data as described above along with a single event condition name and the state of that event condition at the time indicated by the time stamp.

Journal Services

The services available for MMS journals are as follows:

ReadJournal

A MMS client uses this service to read one or more entries from a journal.

WriteJournal

A MMS client uses this service to create new journal entries in a journal. A journal entry can also be created by local autonomous action by the VMD without a client using the WriteJournal service.

CreateJournal, DeleteJournal

A MMS client uses these services to create and delete (if the journal is deletable) journal objects. The CreateJournal service only creates the journal. It does not create any journal entries (see WriteJournal).

InitializeJournal

A MMS client uses this service to delete all or some of the journal entries that are in a journal. For instance, a client can use InitializeJournal to delete old journal entries that are no longer of any interest.

Files

MMS also provides a set of simple file transfer services for devices that have a local file store but do not support a full set of file services through some other means. For instance, many robot implementations of MMS use the file services for moving program (domain) files to the robot from a client application. The MMS file services support file transfer only, not file access. Although these file services are defined in an annex within the MMS standard, they are widely supported by most commercial MMS implementations. The services for files are described below:

FileOpen

A MMS client uses this service to tell the VMD to open a file and prepare it for a transfer.

FileRead

This service is used to obtain a segment of the file's data from a VMD. The client would continue to issue FileOpen requests until the VMD indicates that all the data in the file has been returned. The number of bytes returned in each FileRead response is determined solely by the VMD (and can vary from one FileRead response to the next) and is not controllable by the client.

FileClose

A MMS client uses this service to close a previously opened file. It is used after all the data from the file has been read or can be used to discontinue a file transfer before it is completed.

ObtainFile

A MMS client uses this service to tell the VMD to obtain a file. When a VMD receives an ObtainFile request it would issue FileOpen, FileRead(s) and FileClose service requests to the client application that issued the ObtainFile request. The client would then have to support the server functions of the FileOpen, FileRead, and FileClose services. A third party option is available (if supported by the VMD) to tell the VMD to obtain the file from another node on the network using some protocol (which may or may not be MMS).

FileRename, FileDelete, FileDirectory

These services are used to rename, delete, and obtain a directory of files on the VMD respectively.

MMS Context Management

MMS provides services for managing the context of communications between two MMS nodes on a network. These services are used to establish and terminate application associations and for handling protocol errors between two MMS nodes. The terms *association* and *connection* are sometimes used interchangeably. The node that initiates the association with another node is referred to as the *calling* node. The responding node is referred to as the *called* node.

In a MMS environment, two MMS applications establish an application association using the MMS Initiate service. This process of establishing an application association consists of an exchange of some parameters and a negotiation of other parameters. The exchanged parameters include information about restrictions and attributes that pertain to each node that are determined solely by that node (e.g., which MMS services are supported). The negotiable parameters, on the other hand, are either accepted by the called node or negotiated down (e.g., the maximum message size).

To summarize, the calling application issues an Initiate service request that contains information about the calling node's restrictions and attributes and a proposed set of negotiable parameters. The called node examines the negotiable parameters and adjusts them as necessary to meet its requirements and then returns the result of this negotiation and the information about its restrictions in the Initiate response. Once the calling node receives the Initiate confirmation, the application association is established and other MMS service requests can then be exchanged between the applications.

Once an application association is established either node can assume the role of client or server independent of which node was the calling or called node. For any given set of MMS services, one application assumes the client role while the other assumes the role of the server or VMD. Whether or not a particular MMS application is a client, server (VMD), or both is determined solely by the developer of the application.

MMS Associations

In a connection oriented environment such as OSI or TCP/IP[2], the MMS Initiate service is used to signal to the lower layers that a connection must be established. The Initiate service request is passed through the layers as each layer goes through its connection establishment procedure until the Initiate indication is received by the called node. The connection does not exist until after all the layers in both nodes have completed their connection establishment procedures and the calling node has received the Initiate confirmation.

In a connectionless environment, it is not strictly necessary to send the Initiate request before two nodes can communicate. In an environment where the Initiate service request is not used before other service requests are issued by a MMS client to a VMD, each application must have all the knowledge regarding the other application's exchanged and negotiated parameters using some local means (e.g., a configuration file). This foreknowledge of the other MMS applications's restrictions is the application association from the MMS perspective. Whether an Initiate service request is used or not, application associations between two MMS applications must exist before communications can take place. In some connectionless environments such as MiniMAP, MMS nodes still use the Initiate service to establish the application association before communicating.

Context Management Services

See Module 3

Initiate

This service is used to exchange and negotiate the parameters required for two MMS applications to have an application association.

Conclude

A MMS client uses this service to request that a previously existing application association be terminated in a graceful manner. The conclude service allows the server to decline to terminate the association due to ongoing activities such as a download sequence or file transfer.

Abort

This service is used to terminate an application association in an ungraceful way. The server does not have the opportunity to decline an abort. An abort may result in the loss of data. Although the MMS standard does not provide any protocol for an Abort service, most MMS implementations use other network services for providing this service.

Cancel

A MMS client uses this service to cancel an outstanding MMS service request (e.g. TakeControl) that has not yet been responded to by the server.

Reject

A MMS client or server uses this service to notify the other MMS application that it had received an unsupported service request or a message that was not properly encoded.

[2] While there are profiles of OSI and TCP/IP network communications that do not require connections and can be used with MMS, most commercial MMS implementations for seven-layer LAN environments run over network stacks that require connections.

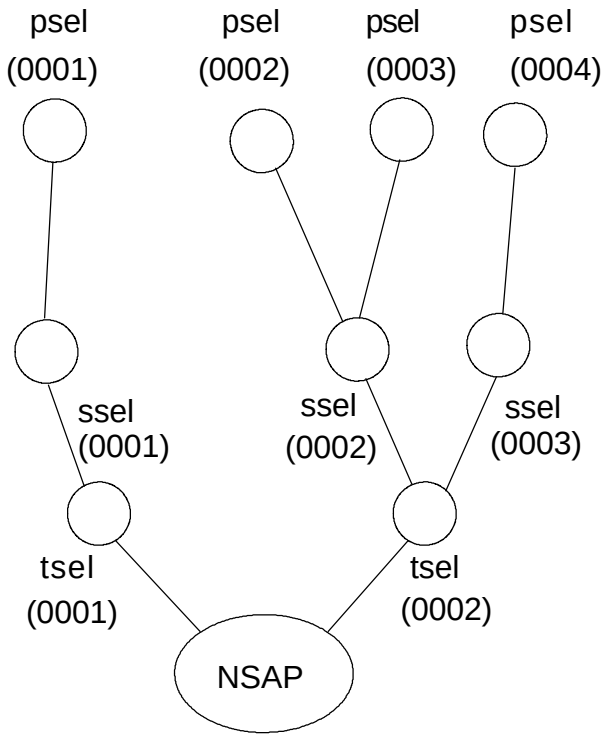
3. Addressing Issues

This section explains the OSI addressing scheme from the network and programmatic points of view. The discussion presented here assumes that you possess cursory knowledge of the OSI 7-layer model.

Presentation Address

In order for one MMS application to establish a connection to another MMS application, the initiating node must know the address of the node with which it intends to communicate. This address is called the Presentation Address and is comprised of the Presentation Selector, Session Selector, Transport Selector, and the Network Address (NSAP).

Selectors are octet strings that can be assigned arbitrarily. The NSAP identifies the node on a OSI network and is equivalent to the IP Address on a TCP/IP network. The sequence of selectors and NSAPs defines a branch through the OSI Stack address tree as shown in the diagram below.



To be able to use the OSI stack, the application must first bind to it. This is done by activating a Presentation Address which creates a new branch (path) in the address tree. Once the presentation address is successfully activated, the application is ready to send and/or receive connect requests using the newly activated path.

When a MMS application attempts to establish a connection, it issues a connect request that contains the presentation address of the application intended to receive the request. First, in order for the request to reach the destination, the specified NSAP must match the NSAP of the remote node. If this is the case, the request will reach its destination. Upon receipt of the connect request, the remote OSI stack examines the selectors present in the request. If it finds a branch in its address tree with matching selectors, the connect request is passed to the application that activated the branch. Otherwise, the connection is rejected.

Figure 14: Selector Diagram

ACSE Parameters

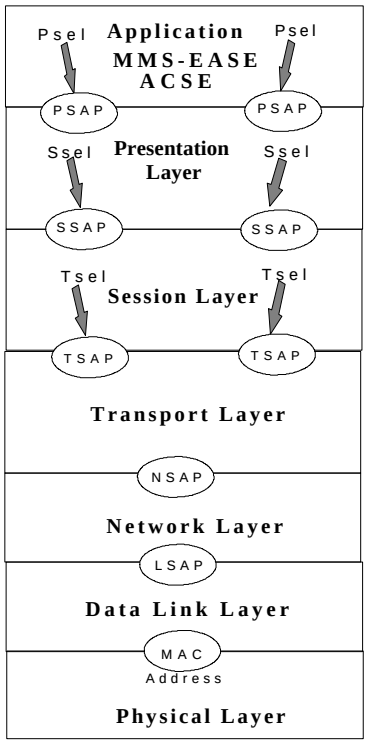
In some situation, as in the case of secure systems, it may be necessary for the called application to know the identity of the caller. In others, as in the case of MMS-based gateways, the identity of the destination must be known. To provide and/or examine such information, ACSE offers a set of parameters that the application can send and receive. These parameters are **A**pplication **P**rocess Title (AP Title), **A**pplication **Entity** Qualifier (AE Qualifier), **A**pplication **P**rocess Invocation Identifier (AP Invoke ID), and **A**pplication **E**ntity Invocation Identifier (AE Invoke ID). These parameters are optional and their use is strictly application-specific. See the "OSI Addressing" section for more information on the format of the ACSE parameters.

AR Names

MMS-EASE defines the term "Application Reference Name", or "AR Name". An AR Name is an ASCII string of up to 32 characters that is used to collectively identify Application Entity information (AP Title and AE Qualifier) and the Presentation Address associated with an application. In other words, an AR Name is not something that is exchanged between two applications over the network, but rather a human-readable shorthand for the ACSE and addressing information that it represents. MMS-EASE applications use AR Names when calling MMS-EASE Connection Management APIs.

Addressing Concepts

This section gives is intended to give you an insight into how OSI addressing works. It introduces new concepts and abstractions that are not required in order to program MMS-EASE application or to configure a OSI Directory Information Base (DIB).



Review the diagram of the OSI stack on the left. Oval accesses shown between the adjacent layers represent the **S**ervice **A**ccess **P**oints (or SAPs). A SAP can be viewed as a channel through which one layer provides a set of services (such as connection management and data transfer) to the layer immediately above it. A SAP is similar to a TCP/IP socket as defined by BSD (Berkeley Software Distribution).

Additionally, the diagram shows Selectors which are represented by arrows. A Selector is an octet string used to identify a specific SAP at a specific layer for all upper layers (Transport and above). Since in most OSI implementations all SAPs at any layer provide the same set of services to their users, all such SAPs are equivalent. Even though theoretically, there may be several selectors that identify a single SAP, for the sake of simplicity, we will assume that each unique selector identifies a unique SAP. Also, since selectors are simply identifiers, their values can be chosen arbitrarily. From this discussion it is easy to see that a OSI selector is functionally equivalent to a TCP/IP Port number, with the exception that the TCP/IP Port number almost always defines an application protocol.

At the Network layer, no selectors are used because the NSAP (or Network Address) itself is used to identify a node on a OSI network, in the same fashion the IP Address is used to identify a node a TCP/IP network.

Figure 15: OSI Seven Layer Stack

The OSI address at any specific layer is constructed by combining all the selectors up to that layer. MMS uses presentation addresses to set up communication channels between two MMS-enabled applications. A presentation address is composed of the Presentation Selector, Session Selector, Transport Selector, and one or more Network Addresses (NSAPs). Note that in most instances, you will only have a single NSAP associated with a presentation address. However, if your system is attached to more than one subnetwork, it is possible to assign a unique NSAP for each such subnetwork.

To summarize, the Presentation Address represents a conduit through an OSI Stack that an application uses to communicate with other OSI-enabled applications on the network.

When communicating with another application process on an OSI network using MMS-EASE, you specify the AR Name configured in the local DIB to represent the remote application. MMS-EASE extracts the presentation address and ACSE parameters from the DIB and passes this information to the OSI stack.

Once the remote address is known, it becomes the job of the Network layer to find the proper subnetwork and the best path to reach the destination NSAP. This process, which consists of associating MAC addresses with NSAPs, is called network routing and is implemented using the ISO End System to Intermediate System (ES-IS) protocol.

Network routing is accomplished in one of two ways: dynamic or static. Dynamic routing occurs when MAC addresses associated with NSAPs are discovered dynamically. With static routing, the user creates static routing records in the Network layer by providing the required information via OSI stack configuration commands. Static routing should only be used in those situations where the remote node does not implement ES-IS. Consult the documentation that accompanies the OSI stack for more information on how to create static routing records.

Naming Authority

In most applications where the network is local, the assignment of SAPs and selectors can be rather arbitrary, and is usually done on an adhoc basis. However, the assignment of names and addresses can be an important issue when designing a large open network. This is where the naming authority comes into the picture. The term “Naming Authority” represents a person or an organization responsible for the assignment of network addresses and ACSE parameters across the network or networks.

OSI Addressing

A summary of the OSI addressing elements is shown below.

AP Title

The AP Title is an **Object Identifier**, assigned by the network naming authority, representing the Application Process Title for your particular application process. An Object Identifier is a sequence of integer values representing an application in the OSI Model. The first value in the AP Title should be 1 (ISO) (the valid range is 0 to 2). The second value should be 1, 2, or 3 (the valid range is 0 to 39). The third value is open for international assignment. If your addressing is local, then this value should be 9999. The rest of the values are arbitrary. For example, values could be assigned to represent type and location of the application.

AP Invoke ID

This is an optional integer value used to identify an Invocation Instance of the Application Process. The use of this parameter is currently not well defined and therefore is discouraged.

AE Qualifier

This is an optional integer value used to qualify the Application Entity.

AE Invoke ID

This is an optional integer value used to identify an Invocation Instance of the Application Entity. The use of this parameter is currently not well defined and therefore is discouraged.

AE Title

Even though an AR Name may be configured to use any or all of the ACSE parameters defined above, the recommended approach is that if ACSE information is required it should be in the form of a AE (Application Entity) Title. The AE Title is comprised of an AP Title and an AE Qualifier. If used, the AE Title should be registered with the appropriate naming authorities for your network.

Presentation Selector

This octet string represents the **P**resentation **Sel**ector used to identify a Presentation SAP and can be up to 16 octets in length.

Session Selector

This octet string represents the **S**ession **Sel**ector used to identify a Session SAP that can be up to 16 octets in length.

Transport Selector

This octet string represents the **T**ransport **Sel**ector used to identify a Transport SAP that can be up to 32 octets in length.

Network Address (NSAP)

The **N**etwork **S**ervice **A**ccess **P**oint (NSAP), as defined by ISO, is an octet string that represents the network address for a given node on the network. A NSAP can be 1 to 20 octets in length and is comprised of two parts: the Initial Domain Part (IDP) and the Domain Specific Part (DSP). The DSP may contain subfields whose structure is defined by the naming authority responsible for administering the IDP.

The Initial Domain Part (IDP)

The IDP identifies the naming authority, such as ANSI and NIST, responsible for managing an NSAP space. Such authorities have been allocated IDP values by ISO that they can use to distribute NSAP addresses to organizations under their control.

The IDP consists of two fields: the AFI and the IDI. The authority and format identifier (AFI) identifies the type of address used in DSP. The initial domain identifier (IDI) determines which domain the DSP is part of.

AFIs are assigned by ISO IS 8348: *Information processing systems — Data communications — Network service definition* — Addendum 2: *Network layer addressing*. The AFI indicates the format for the remainder of the network address and is represented as one byte. Table I shows some selected AFIs.

AFI (in hex)	Author
0x39	ISO DCC
0x41	F.69
0x47	ISO 6523-ICD
0x49	LOCAL

Table I: Selected AFIs

ISO DCC IDI format (0x39)

The IDI consists of a three-digit numeric code allocated according to ISO 3166.

F.69 IDI format (0x41)

The IDI consists of a telex number of up to 8 digits, allocated according to CCITT Recommendation F.69, commencing with a 2 or 3-digit destination code.

ISO 6523-ICD format (0x47)

The IDI consists of a 4-digit International Code Designation (ICD) allocated according to ISO 6523.

Local IDI format (0x49)

The IDI is Null.

The Domain Specific Part (DSP)

The DSP is subdivided into a number of fields, each with its own meaning, that allow to uniquely identify an open system on the network and provide necessary routing information.

At this point, we will define some terms that will be used throughout our discussion:

End System - an open system that support OSI layers 1 through 7 and is capable of initiating or responding to communications.

Intermediate System - a OSI Network layer router.

Subnetwork - a collection of network equipment used to interconnect open systems.

Area - a set of end systems and intermediate systems interconnected by one or more subnetworks.

Routing Domain - a set of end systems and intermediate systems interconnected by one or more subnetworks that have been grouped together by way of some routing mechanism.

DSP Format Identifier (DFI) - a field in the DSP used to identify a DSP format.

Reserved field - a portion of the DSP currently not in use.

NSAP Selector (Nsel) - the right most octet in the DSP that identifies the entity attached to the network layer. The recommended value for the Network Selector is 01.

In order to provide for ISO Intermediate System to Intermediate System (IS-IS) routing and End System to Intermediate System (ES-IS) routing, the DSP will contain one or more of the following fields:

- Routing Domain Identifier
- Area Identifier
- Subnetwork Identifier
- End System Identifier

ISO defines the following NSAP routing model:

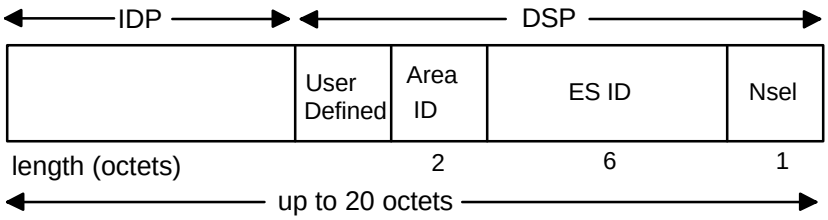


Figure 16: NSAP Routing Model

NSAP Formats

This sections lists the most widely used NSAP formats.

ANSI Format

Under the ISO DCC AFI (0x39), the US is assigned the Data Country Code of 0x840 (padded with 0xF) which is administered by ANSI. ANSI has designated the first three octets of the DSP to be an Organization Identifier, with the remaining 14 octets to be defined by each organization.

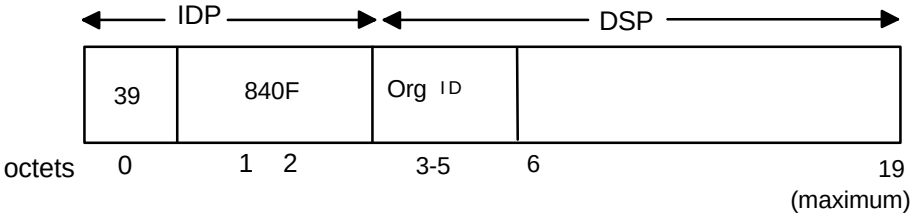


Figure 17: ANSI Format

US GOSIP Format

The US GOSIP format is administered by NIST.

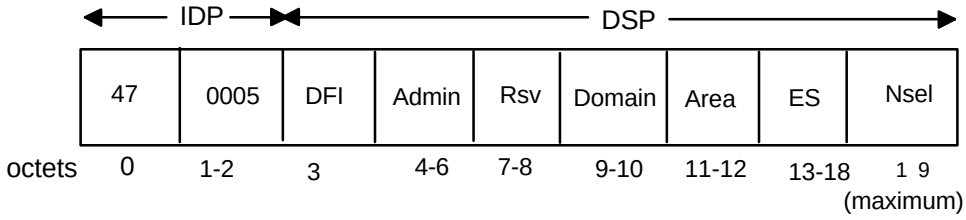


Figure 18: US GOSIP Format

The definitions of the NSAP fields are as follows:

DFI - DSP Format Identifier. See above for more information.

Admin - equivalent in functionality to Org ID.

Rsv - reserved field.

Domain - the Routing Domain containing this NSAP.

Area - the Area containing this NSAP.

ES - an End System identifier, unique within the containing Area.

Nsel - identifies the Network Service user. US GOSIP recommends that the value of 01 be used for the Transport protocol.

ISO Local Format

This format is intended for use on isolated networks. In the local format, the IDI is null.

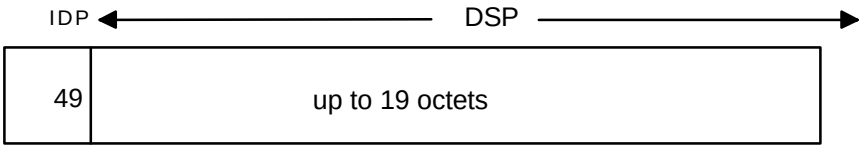


Figure 19: ISO Local Format

Conclusion

In conclusion, it is important to mention that when you have to define part of the DSP, as in the case of the ANSI format or the local format, you should provide the NSAP structure shown below. This will ensure that your format can be easily migrated to other formats and can support IS-IS routing.

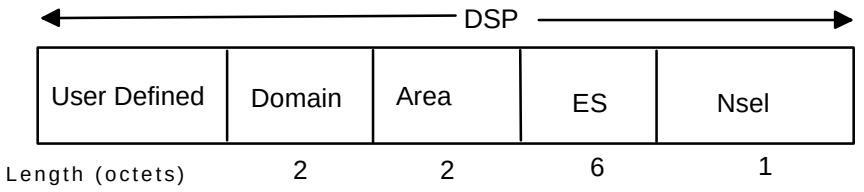


Figure 20: Default NSAP Format

The definitions of the fields are as follows:

Domain - the Routing Domain containing this NSAP.

Area - the Area containing this NSAP.

ES - an End System identifier, unique within the containing Area.

Nsel - a network selector. The recommended value is 01.

Context Negotiation

Because MMS runs on top of ACSE/Presentation, there is another issue to consider regarding establishing a communications path between MMS applications. This additional consideration is Presentation Context and Application Context negotiation.

A Presentation Context is represented by an Abstract Syntax and a set of one or more Transfer Syntaxes. An Abstract Syntax identifies the semantics of the language or languages that the application can understand. Transfer Syntaxes identify the transfer (encoding) mechanisms that the application can use to deliver the protocol data to its peer. Conceptually, an Abstract Syntax is analogous to the C language and Transfer Syntaxes are analogous to the machine code generated by various C compilers.

An Application Context identifies the type of protocol the application supports.

Context negotiation takes place during association phase. At this time, the initiating application proposes an Application Context and a set of Presentation Contexts. The responding node examines this information, decides whether the Application Context is compatible, selects which Presentation Contexts it is willing to support and for each Abstract Syntax chooses a Transfer Syntax that it prefers to use. This completes context negotiations.

Abstract Syntaxes, Transfer Syntaxes, and Application Contexts are represented by Object Identifiers.

The following sections give the Object Identifiers for Abstract Syntaxes, Transfer Syntaxes, and Application Contexts used by MMS.

MMS Abstract Syntaxes

The Object Identifier for the MMS Core Presentation Abstract Syntax is {1 0 9506 2 1}.

Additional Object Identifiers are assigned for the abstract syntaxes corresponding to the various MMS companion standards.

ASN.1 Transfer Syntax

The only transfer syntax currently supported by MMS is the ASN.1 Basic Encoding Rules (ISO IS 8825). The Object Identifier for the ASN.1 is {2 1 1}.

MMS Application Contexts

Currently MMS identifies two Application Contexts: MMS International Standard (IS) {1 0 9506 2 3} and MMS Draft International Standard (DIS) {1 0 9506 1 1}. According to the MMS Implementors' Agreements, the Application Context should be ignored. Therefore, your application need not be concerned with setting this parameter when establishing an association. Instead, the MMS Version number, sent in the MMS Initiate request and response primitives, determines whether the communicating parties prefer to use IS (Version 1) or DIS (Version 2).

Supported Stack Profiles

The following sections give a brief overview of the OSI and non-OSI stack profiles supported by MMS-EASE.

Network Based Profiles

These profiles are supported on LANs, such as Ethernet (IEEE 802.3) and FDDI (IEEE 802.6), and WANs, such as X.25.

OSI

This is the standard profile that supports the OSI 7-layer model. In this configuration, the OSI stack is comprised of ACSE, Presentation, and Session running over Transport Class 4 with connectionless Network that implements routing using ES-IS.

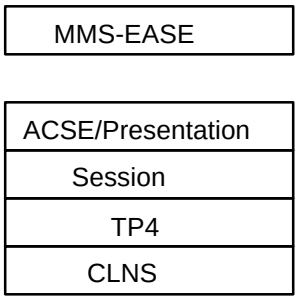


Figure 21: Standard OSI Profile

RFC1006

RFC 1006, "ISO Transport Services on Top of the TCP", specifies a standard protocol that allows OSI upper layers (ACSE, Presentation, and Session with Transport Class 0) to run on top of the TCP/IP stack. In effect, this allows you to run MMS applications over existing TCP/IP-based LANs and WANs. This approach also replaces the task of administering OSI NSAPs with a simpler task of administering IP Addresses.

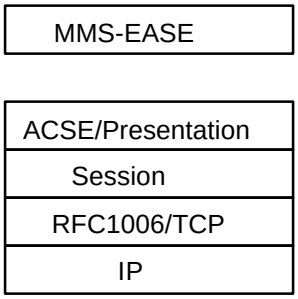


Figure 22: RFC 1006 Profile

RS-232 Based Profiles

UCA^{tm} Reduced Stack

The UCA Reduced Stack profile allows MMS to be used over a modified version of HDLC called Asynchronous Data Link Control (ADLC). This profile allows you to use MMS in Supervisory Control and Data Acquisition (SCADA) networks that are based upon Radio or modem point-to-point or point-to-multipoint topologies.

As the name implies, this profile does not include implementation of all 7-layers of the OSI model, most visibly the network layer. Functionally, the lack of a network layer means that this profile is not routable and is therefore limited to a single network (RS-232) segment.

UCA Trim-7

As the Reduced Stack profile, the UCA Trim-7 profile allows MMS to be used over a modified version of HDLC called Asynchronous Data Link Control (ADLC). This profile allows you to use MMS in Supervisory Control and Data Acquisition (SCADA) networks that are based upon Radio or modem point-to-point or point-to-multipoint topologies.

As the name implies, this profiles implements OSI upper layers (Session and Presentation) more efficiently than the standard OSI profile by using much smaller PDU headers. The efficient upper layers are not interoperable with the standard OSI profile.

Unlike the Reduced Stack profile, the network layer is active and therefore, this profile is capable of routing and is not limited to a single network (RS-232) segment.

4. Directory Services - X.500

Introduction

The task of administering OSI addresses and AR Names can be very complex. In order to simplify this task and provide for a centralized database containing addressing and other information that can be reached by all nodes on the network, the X.500 Directory Services can be employed.

The X.500 series of specifications defines a distributed Directory Service that enables users and applications to communicate with one another. Directory Service facilitates the mapping of user-friendly location-independent names to more difficult to remember information such as network addresses, telephone numbers, or personal addresses. These user-friendly names are called Directory **D**istinguished **N**ames, or DNs. A DN is a sequence of **R**elative **D**istinguished **N**ames (RDN) that unambiguously identifies a Directory entry.

Directory Service acts on a Directory-specific database, called the **D**irectory **I**nformation **B**ase (DIB), hierarchically structured according to the **D**irectory **I**nformation **T**ree (DIT). Each entry stored in the DIT represents an object, such as a person, an organization, an application process, or an application entity. Each entry consists of the object's name and all of its related information, represented as a set of attributes. Each attribute is defined by its type and one or more values. The set of definitions and constraints concerning the structure of the DIT and the possible ways entries can be named, the information that can be held in an entry, and the attributes used to represent that information is called the Directory Schema.

The DIB is globally distributed over the network, partitioned in subtrees, with each subtree under the responsibility of a particular Directory server, called the **D**irectory **S**ystem **A**gent (DSA). A number of such servers communicate with one another to provide Directory services to Directory clients. The Directory clients, such as users and application processes, have at their disposal a **D**irectory **U**ser **A**gent (DUA) for requesting Directory services and managing Directory protocols.

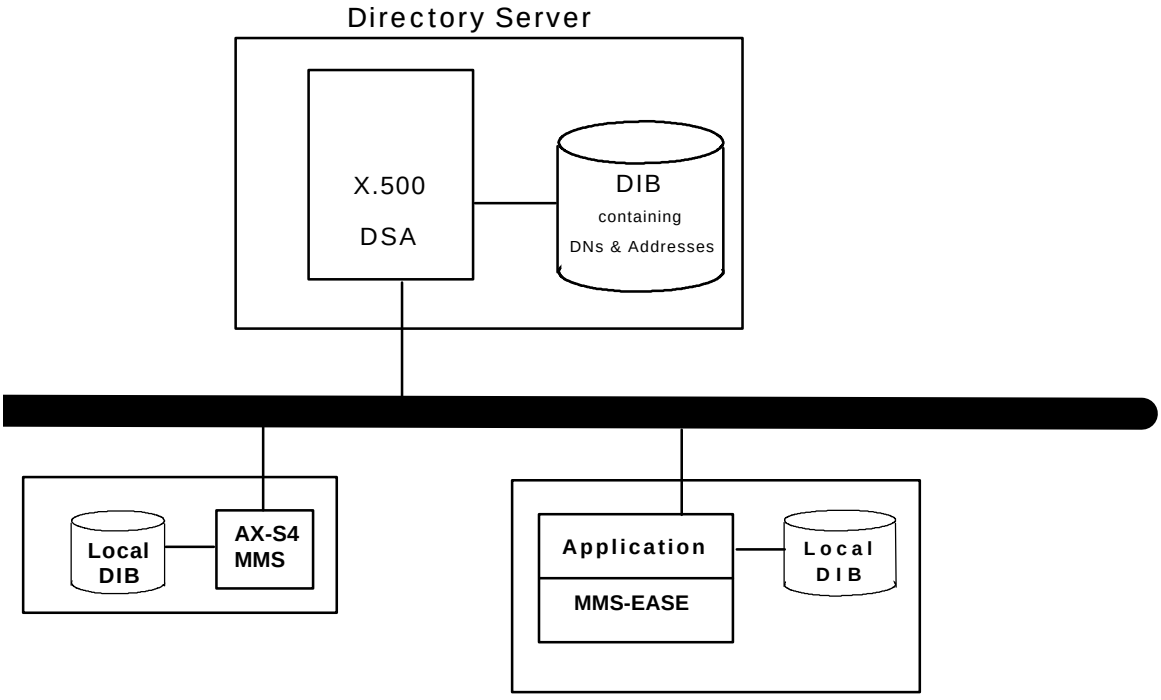


Figure 23: Diagram of Directory Services

The protocol between a DUA and a DSA is called **D**irectory **A**ccess **P**rotocol (DAP). The protocol between two DSAs is called **D**irectory **S**ystem **P**rotocol (DSP).

DAP is a very complex protocol that requires the services of OSI upper layers, such as ROSE, ACSE, Presentation, and Session. While building a complex DSA process running on a powerful workstation may not present a problem, having this complexity in a client may be more troublesome. In view of this problem, another protocol was designed and originally defined in RFC1487 that has now been made obsolete by RFC1777. The new protocol, called **L**ightweight **D**irectory **A**ccess **P**rotocol (LDAP), provides access to the X.500 Directory without the complexity inherent in DAP. Initially, LDAP servers acted as gateways between LDAP clients and DSAs or as front-ends for DSAs. Currently many vendors offer stand-alone versions of LDAP servers. These servers manage the Directory themselves and do not require a DSA.

Both protocols offer the same set of services, such as Read, Modify, Search, List, that allow the user to create, read, modify, and delete the Directory entries, as well as perform the Directory searches and list the Directory entries given certain search criteria.

The SISCO OSI Stack and MMS-EASE support both DAP and LDAP. It is up to you to choose whether to use full X.500 (DAP), LDAP, or both, based on your personal preference and your system requirements. If you decide to enable both protocols, MMS-EASE will first attempt to perform name resolution using X.500, and then, in case of failure, will route the request to a LDAP server.

The Structure of a Distinguished Name

As mentioned previously, a Distinguished Name is a sequence of Relative Distinguished Names. A RDN is represented by a set of **A**ttribute **V**alue **A**ssertions (AVA). An AVA is defined by an attribute type and an attribute value. The data type of an attribute value depends entirely on the syntax of the attribute for which the attribute value is defined.

The meaning of these terms will become more evident in the following sections.

String Representation of Distinguished Names

To retrieve the information stored in a Directory entry, the DUA must know the entry's Distinguished Name (DN). You will provide a DN using either of the two DN formats supported by MMS-EASE. One format offers OSF-style syntax and will be referred to in this document as the X.500 DN format. The other format follows the rules for the string representation of Distinguished Names as defined in another LDAP specification, RFC1779 and will be referred to as the LDAP DN format.

X.500 DN Format

This format uses the top down approach for specifying a sequence of RDNs to form a DN. For example:

/Country=US/Organization=SISCO/OrganizationalUnit=Engineering/CommonName=NTServer

- Each component in this sequence such as /Country=US or /CommonName=NTServer is an RDN.
- Each component within an RDN such as Country=US is an AVA.
- The value on the left side of the equals sign is an attribute type. The value on the right side is an attribute value.

The following abbreviations can be used for attribute types to simplify the task of constructing a DN:

CN	**C**ommon**N**ame
L	**L**ocality
ST	**St**ateOrProvince
O	**O**rganization
OU	**O**rganizational**U**nit
C	**C**ountry
STREET	**Street**Address

Using these abbreviations, the previous example could be re-written as follows:

/C=US/O=SISCO/OU=Engineering/CN=NTServer

Note: Attribute types are not case sensitive. Attribute values, on the other hand may be case sensitive depending on the attribute syntax.

If an RDN has two or more AVAs, use the '&' character to separate the AVAs. For example:

/C=US/O=SISCO/OU=Engineering&L=Michigan/CN=NTServer.

If an attribute value contains special characters such as '=', '/', '&', '\', or white space characters, they must be escaped with the '\' character, as in the following example:

/C=US/O=SISCO/OU=R\&R/CN=Golf\ Course

LDAP DN Format

RFC1779 which defines the string representation of a Distinguished Name per LDAP, unlike OSF, uses the bottom up approach for specifying a DN. Also, it uses the comma instead of the forward slash to separate RDNs. The example from the previous section would look as follows:

cn=NTServer, ou=Engineering, o=SISCO, c=US

Note: You can still use the same set of abbreviations for attribute types as provided in the previous section.

If you have a multi-component AVA, you need to use the '+' sign to separate the AVAs. For example:

cn=NTServer, ou=Engineering + l=Michigan, ou=SISCO, c=US

If you need to include a special character in an attribute value, such as '=', '+', ',', '"', you can either embed the attribute value in double quotes or escape the special character with the backslash '\'.

Naming Context

In order to shorten DN strings, MMS-EASE offers the Naming Context feature. Naming Context is a partial DN string applied to another user-supplied partial DN string to form a complete DN. The following example explains how Naming Context works:

Naming Context = /C=US/O=SISCO

User-supplied DN= ./OU=Engineering/CN=NTServer

DN = /C=US/O=SISCO/OU=Engineering/CN=NTServer

Naming Context can be specified using either X.500 format (prefix) or LDAP format (suffix). In the case of LDAP, Naming Context in the example above would look like the following:

Naming Context = o=SISCO, c=US

Naming Context will be applied to a DN in the following cases:

A DN is given in X.500 format and the first character is the period, as in the example above.

A DN is embedded in round parentheses. For example:

(cn=NTServer, ou=Engineering) or (/OU=Engineering/CN=NTServer).

MMS-EASE AR Names

If you pass an AR Name that uses DN format to any of the MMS-EASE API functions that require an AR Name as one of the input parameters, MMS-EASE will assume that the AR Name is a DN and will attempt to resolve it using the Directory Read service. In all other cases, the AR Name will be assumed to have been defined in the local DIB.

Note: Locally defined AR Names may not contain the following special characters: ‘/’, ‘\’, ‘=’, ‘+’, ‘&’, ‘.’, ‘,’, white space characters.

Authentication

MMS-EASE supports simple Directory authentication. This means that you can instruct MMS-EASE to provide your name and password when it binds to the Directory. See page 1-90 for information on the functions used to accomplish this.

If these functions are called before **`strt_MMS`**, MMS-EASE will relay authentication information to the Directory server. You do not have to call both functions.

Note: With LDAP servers, even though your authentication may fail, you will still be able to perform Read operations on the Directory. This is not the case with X.500. If you are not authenticated, binding will be rejected and the association terminated.

System Implementation Notes

In this section we will discuss the actions that will be required of the System Designer, the DSA Administrator, and the MMS-EASE programmer in order to successfully design, implement, and use a distributed Directory environment.

The vast majority of MMS-based applications using the Directory will require that it make available the addressing information (Presentation Address and AE Information) for all the nodes in the system whose addresses cannot or should not be stored in the application's local DIBs. The System Designer will need to determine which addresses should be stored in the Directory and which addresses should be stored in the local DIB.

Once the System Designer makes the determination and relays this information to the MMS-EASE programmer, the programmer will add the addressing information to be stored in the local DIB in the form of AR Names.

For those nodes whose addressing information is to be stored in the Directory, the System Designer will provide the DSA Administrator the addressing information and the node description. The DSA Administrator will need to determine where in the distributed DIT this information should reside. The Administrator will:

1. Designate a branch in the DIT to store such information.
2. Create entries of object class applicationProcess to represent application processes in the system.
3. Create one or more sub-entries of object class applicationEntity to represent application entities within an application process for each application process entry. These are the Directory entries that will contain Presentation Address and, optionally, AP Title and AE Qualifier.
4. Make the necessary modifications to the Directory Schema, if AP Title (Object Identifier) and AE Qualifier (Integer) are required, such that the object class applicationEntity may contain these attributes. Procedures for modifying the Directory Schema are described in the manuals that accompany the Directory Server product.
5. Decide on the level of authentication and access control that is appropriate for those applications that will require access to the Directory to retrieve this information. Again, the Directory Server manuals will provide the information on how to assign access control levels and set up user authentication.

The Administrator will at this point relay all the relevant information to the System Designer. This information will include:

1. The Directory Server's network address: Presentation Address in the case of the DSA, IP Address and TCP/IP port number in the case of the LDAP Server.
2. The DSA's AP Title and AE Qualifier, if required.

3. Authentication information (Sets of Distinguished Names and Passwords) for use by the Directory users, if required.
4. The Distinguished Name of the Directory entry under which application process entries reside.
5. Distinguished Names for all application entity entries.
6. The schema requirements for allowing to store AP Title and AE Qualifier in an application entity entry.

The System Designer will pass on all or part of this information to the MMS-EASE programmer. Using this information, the programmer will:

1. Configure the DUA and DSA and/or LDAP portions of the local configuration per the instructions described in individual product's Installation and Configuration Guide.
2. Set up the Naming Context.
3. Make modifications to the local directory schema per the procedure described in the section titled "**Directory Schema**" in the Installation and Configuration Guide for your MMS-EASE product, if such modifications are necessary to align the local schema representation with the schema defined on the Directory Server.
4. Use the Authentication information to program the call to **`mds_set_def_userid`** and/or **`ldap_set_def_userid`**. See page 1-90 for more information on these functions.
5. Use Distinguished Name information to program the MMS-EASE API such as **`mv_init`**, **`mllp_act_ar_name`** and **`mllp_act_arname`**. See pages 1-0 and 1-0200 for information on these function.

5. MMS-EASE Application Program Interface

MMS-EASE is an **E**mbedded **A**pplication **S**ervice **E**lement for **MMS** that meets the requirements for OSI communications between programmable devices on the factory floor. It consists of a library of 'C' language function calls and data structures that provide application programs access to all the services needed for communications using the MMS protocols in a way that is independent of any particular OSI interface board or operating system. MMS-EASE has all the functions needed for most applications using such MMS services as variable, context, domain, file, and semaphore.

MMS-EASE is written in modular fashion, with each part using the services of the others. Although most of these internal interfaces are exposed, most applications do not need to deal with them. Almost everything needed by a MMS application is provided directly by the MMS-EASE libraries. The user writes the application program, compiles it, and then links the user object code with the MMS-EASE libraries to create an executable MMS compatible application.

The MMS-EASE **A**pplication **P**rogram **I**nterface (API) consists of two different types of interfaces allowing a choice in the manner that actions are taken by the application program. The **P**aired **P**rimitive **I**nterface (PPI) can be used to take all actions within your program or the **V**irtual **M**achine **I**nterface (VMI) can be used to let the virtual machine take the actions.

When the local node, containing the user's program, requests a service of another node or when another node requests a service of the local node, some action (besides the communications functions), must be taken by the local node. In some cases, there are two choices that can be taken. For example, a file transfer can be accomplished in one of two ways:

> The local node receives an indication, from a remote node, to open a local file. Besides receiving the indication and sending of a response, the local file must be physically opened. This would involve some type of interaction with the local operating system.
>
> Using PPI means that you would have total control over file handling. It would not be opened automatically by the virtual machine. PPI would be used to send a response, by calling the appropriate primitive response function after the local program actually opened the file.
>
> If you did not want to deal with the details of opening files and keeping track of open files, the VMI could be used to open the file **AND** send the response, simply by calling the appropriate virtual response function.
>
> Many of the virtual machine support functions (used to define and manipulate MMS objects) can also be used for PPI calls to simplify the processing required.

In the case of a variable read:

> The local node wants to read a variable from a remote node.
>
> If issuing the read request using a PPI function, your application program is responsible for interpreting the confirmation, and writing the data received into a local variable, if needed.
>
> If using a virtual request function, only the channel number to use, the remote variable's name, type, and a pointer to where the data is to be placed locally needs to be supplied for the request. The VMI takes care of sending the request, receiving the confirmation, and, then, writing the data into local memory as specified.

In general, it is advantageous to use the VMI because it relieves you from having to deal with the actions dictated by the service being performed. In addition, the VMI also provides higher level functions to be executed without any additional programming effort. For instance, the MMS specification only allows a single segment of a file to be read for any given request. The VMI allows reading multiple file segments, and automatically issues multiple requests until all the file segments specified have been read.

Paired-Primitive Interface (PPI)

The **P**aired **P**rimitive **I**nterface (PPI) of MMS-EASE allows sending requests and responses and receive confirmations independently of the action(s) required by the application program to service the requests.

However, despite the name, the PPI is more than just a primitive level interface. It also ”pairs“ confirmations to responded requests. MMS-EASE accomplished this by returning a pointer to the data structure used to initiate the request when the confirmation to the request is received. This added level of functionality relieves the application programmer from having to determine which requests match to the corresponding received confirmations. MMS-EASE also takes care of handling the InvokeIDs required by the MMS specification. MMS-EASE automatically assigns the InvokeIDs, and provides them to your program.

Application of the PPI

The PPI is especially useful for applications where you need to have total control over the data sent in the MMS messages or the actions taken. Because MMS is designed to be very general purpose and able to address the widest variety of applications possible, it contains much more than what the typical application needs. The virtual machine, on the other hand, is designed to automatically perform those certain operations that most applications would have to do anyway. However, there may be applications where the data elements or operations supported by the virtual machine are not sufficient. In this case, the PPI would be used. The PPI supports ALL the functionality of the MMS services. Another application of the PPI may be where the host computer on which MMS-EASE executes is not the destination device. In this case, an external device (e.g., robot) is the actual destination device. The defined domains, files, etc., actually reside in the robot system, not necessarily on the host system executing MMS-EASE. However, the virtual machine does provide some ”hooks“ to deal with external objects if the user prefers the simplicity of the virtual machine over the flexibility of the PPI.

NOTE: Use the PPI or the VMI consistently. For instance, if the VMI is used to open a file, the VMI also should be used to close that file. If not, the virtual machine will still think the file is open. It may cause unnecessary errors if a remote node tries to close the file again.

Virtual Machine Interface (VMI)

The MMS-EASE **V**irtual **M**achine **I**nterface (VMI) provides MMS operation processing capabilities, in addition to the communication functions provided with the Paired Primitive Interface. Operation processing is an essential part of a network member; that is, the communication facilities of MMS are of no value without the associated actions. The VMI provides the required operations where feasible. The VMI is also flexible enough to allow variables physically resident on an external device to be handled by the virtual machine. This is true, even though these variables do not exist permanently within the host environment.

VMI Requests

Virtual machine requests are initiated by function calls containing all the required parameters for the operation. For most virtual machine operations, Operation-Specific data structures do not have to be provided as in the PPI functions. Instead, VMI request functions pass data directly through the function call, and only the information required for the function is needed.

VMI Responses

The virtual machine also may be used by the application to respond to some indications. These functions are invoked by calling the appropriate **mv_xxxx_resp** function with a pointer to the indication control data structure as an argument. These response functions typically take care of all the actions required by the indication automatically, such as opening a file, or reading a variable. This alleviates having to deal with the indication at the primitive level. However, the **mv_xxxx_resp** functions typically do not support all possible options for a given MMS service. If servicing some options not supported by the virtual machine, the request should be handled using the PPI.

6. Function Classes

The MMS-EASE application program interface consists of thirteen separate classes of functions. Each class of functions performs a similar set of actions for each of the MMS service elements supported. A brief explanation of each of the function classes is described below.

1. **PAIRED PRIMITIVE REQUEST FUNCTIONS** allow an application program to issue request primitives to other MMS compatible nodes on the network without invoking any of the virtual machine capabilities. Functions of this class are named: **mp_xxxx**, where "**xxxx**" is specific to the particular service being requested. The contents of these functions are determined by MMS-EASE, and are not user definable. See page 1-57 for further information on primitive request functions.

2. **PAIRED PRIMITIVE RESPONSE FUNCTIONS** allow an application program to issue responses to indications from other MMS compatible nodes on the network. These are used with primitive request functions. It is the responsibility of the application program to perform any actions required, and to format the data structures necessary to generate correctly a response. Functions of this class are named: **mp_xxxx_resp**, where the "**xxxx**" is specific to the particular service (or indication) to which is being responded. The contents of these functions are determined by MMS-EASE and are not user definable. See page 1-57 for further information on primitive response functions.

3. **USER INDICATION FUNCTIONS** are pre-named, user-defined functions called by MMS-EASE when an indication from another node on the network is received. MMS-EASE does not take any action itself, but allows determining whether to let the virtual machine handle the response (by calling the appropriate virtual response function), or to allow the user to take any desired action before calling a primitive response function. Functions of this class are named: **u_xxxx_ind**, where "**xxxx**" is specific to the service requested. The contents of these functions are completely user-defined allowing any action required by the indication to be taken, then issuing the response or allowing the virtual machine to handle the indication. See page 1-58 for further information on user indication functions.

4. **PAIRED PRIMITIVE USER CONFIRMATION FUNCTIONS** are pre-named, user-defined functions called by MMS-EASE when a confirmation to a primitive request function is received. MMS-EASE automatically pairs the confirmations to the responding requests by providing a pointer to the data structure used by the primitive request function. Functions of this class are named: **u_mp_xxxx_conf**, where the "**xxxx**" is specific to the particular service to which has been responded. The contents of these functions are completely user-defined allowing any actions required by the application to be performed. See page 1-58 for further information on primitive user confirmation functions.

5. **ERROR RESPONSE FUNCTIONS** are called by the application program to send error responses to indications. At the PPI level, a function of this class would be used when a user indication function is called, and an error response needs to be sent. The contents of these functions are determined by MMS-EASE and are not user-definable. See page 1-59 for further information on error response functions.

6. **VIRTUAL MACHINE REQUEST FUNCTIONS** allow the application program to issue requests to other MMS compatible nodes on the network, but allow the virtual machine to automatically handle the responses to these requests. For instance, if the **mv_fopen** function is called, the virtual machine will format all the necessary data structures itself, issue the request, and automatically update its internal database when the response to that request comes. The virtual machine functions generally take care of all the details of handling the dialog and performing the necessary actions required to issue requests. Functions of this class are named: **mv_xxxx**, where "**xxxx**" is specific to the service request being made. The contents of these functions are not user definable. See page 1-59 for further information on virtual request functions.

7. **VIRTUAL MACHINE RESPONSE FUNCTIONS** are used to respond to indications received from other nodes on the network using the virtual machine. When using these functions, the application program does not need to perform the actions requested of it by the indication. The virtual machine will take care of it. If the virtual machine's capabilities are not wanted, the appropriate primitive response function should be called instead. Functions of this class are named: **mv_xxxx_resp**, where "**xxxx**" is specific to the particular primitive to which is being responded. The contents of these functions are determined by MMS-EASE, and are not user definable. See page 1-60 for further information on virtual response functions.

8. **VIRTUAL MACHINE USER CONFIRMATION FUNCTIONS** are pre-named, user-defined functions called by MMS-EASE when a confirmation to a virtual request function is received from the remote node. MMS-EASE will have automatically taken any action required to update its internal data base with the results of the request before calling the virtual user confirmation function. Functions of this class are named: **u_mv_xxxx_conf**, where "**xxxx**" is specific to the particular service to which has been responded. The contents of these functions are completely user-defined, allowing the user to perform any actions required by their application not already performed by the virtual machine. See page 1-61 for further information on virtual user confirmation functions.

9. **LLP REQUEST FUNCTIONS** are called by the application program to perform those Lower Layer Provider (LLP) specific functions required by the network that are not directly supported by the specified MMS services. These functions deal with the assignment of process titles to channels and, in general, deal with network addressing. Functions of this class are named: **mllp_xxxx,** where the "**xxxx**" is specific to the particular operation requested. The contents of these functions are determined by MMS-EASE and are not user definable. See page 1-61 for further information on LLP request functions.

10. **USER LLP FUNCTIONS** are pre-named, user-defined functions called by MMS-EASE when confirmations or indications of LLP level primitives, not supported directly by the MMS specification, are received. These functions are used to signal the receipt of association indications and confirmations. Functions of this class are named: **u_mllp_y_xxxx**, where the "**xxxx**" is specific to a particular service and "**y**" indicates the network type. See page 1-62 for further information on user LLP functions.

11. **SUPPORT FUNCTIONS** are called by the application program to perform various support functions required by MMS-EASE. They are used to update such items as the local variable definition tables, and type definition tables. Functions of this class are named: **ms_xxxx**, where "**xxxx**" is specific to the type of support function to be performed. The contents of these functions are determined by MMS-EASE, and are not user definable. See page 1-62 for further information on support functions.

12. **ADDRESS RESOLUTION FUNCTIONS** are user-defined functions called by the MMS-EASE virtual machine when it needs to resolve addresses to variables. These functions are written to resolve addresses for variables that are being accessed over the network. See page 1-62 for further information on address resolution functions. The actual functions are described in **Volume 2 — Module 5 — Variable Access — Address Resolution Functions**.

13. **INTERFACE FUNCTIONS** handle the sending of messages, receiving confirmations and indications, initializing the variable tables. Most of the user-defined functions are called as part of the interface function operation. The contents of these functions are determined by MMS-EASE, and are not user definable. See page 1-62 for further information on interface functions.

Paired Primitive Request Function Class

Paired primitive request functions allow sending MMS request primitives to other nodes on the network without using the capabilities of the virtual machine. Generally, these functions are used to request that the remote node take some form of action as specified by the request.

The primitive request functions take the form:

`MMSREQ_PEND *mp_xxxx (ST_INT chan,` "Operation-Specific data structure" **`*info)`**

Parameters:

"xxxx" See **Volume 1 — Appendix D** for a list of all function names. Examples are:

fopen for a file open request, or *read* for a read variable request.

chan This is the channel number over which the request is to be made.

info This pointer points to the Operation-Specific data structure that contains the data for this request. See the sections on Operation-Specific Data Structures present in all other modules.

In general, when a primitive request function is called, the following actions take place:

1. An InvokeID is determined by MMS-EASE (the user does not assign InvokeIDs).
2. A structure of the type **MMSREQ_PEND** is allocated and initialized.
3. A message (a MMS request PDU) is constructed (encoded) using the information pointed to by ***info**.
4. The appropriate command function is called to send the message to the LLP.
5. Control is then returned to the calling program with a pointer to the **MMSREQ_PEND** structure used. In case of an error, this pointer is set to null and **`mms_op_err`** is written with the error code.

When control is returned to the calling program, it does not necessarily mean that the message has been delivered to the OSI Stack. If an error is later detected, MMS-EASE calls the **u_llp_error_ind** function. This indicates that an error condition has occurred. Alternatively, the channel state information contained in **mms_chan_info[chan].ctxt.xmit_pend**, described on page 1-78, can be used to determine when the message has been delivered to the OSI Stack.

When using the primitive request functions, the Operation-Specific data structures needed by the particular request must be provided by the application.

Paired Primitive Response Function Class

Paired primitive response functions allow sending responses to indications received from another node on the network without using the capabilities of the virtual machine. These functions are well suited to be called from within a user indication function. However, a primitive response function may be called from anywhere within an application program.

The primitive response functions take the form:

`ST_RET mp_xxxx_resp(MMSREQ_IND *ind,` "Operation-Specific data structure" **`*info);`**

Parameters:

"xxxx" See **Volume 1 — Appendix D** for a list of all function names. Examples are:

fopen for a file open response, or *read* for a read variable response.

ind This pointer to the indication control data structure of type **MMSREQ_IND** is passed to the appropriate user indication function, **u_xxxx_ind**.

info If used, this pointer points to the Operation-Specific data structure that contain the data specific to the response to be sent.

In general, when a primitive response function is called, the following actions take place:

1. A message (a response PDU) is constructed (encoded) using the information pointed to by **ind** and **info**.
2. The appropriate command function is called to send the response message to the LLP.
3. Control is returned to the calling program with a return value indicating the success or failure of the function. If failure, an error code is returned.

When control is returned to the calling program, it does not necessarily mean that the message has been delivered to the OSI Stack. If an error is later detected, MMS-EASE calls the **u_llp_error_ind** function. This indicates that an error condition has occurred. Channel state information, contained in **mms_chan_info[chan].ctxt.xmit_pend** described on page 1-78, can be used to determine when the message has been delivered to the lower layers of the network. Operation-Specific information must be committed only until the application returns from the **mp_xxxx** function.

User Indication Function Class

User indication functions are pre-named, user-defined functions called by MMS-EASE when a request from a remote node is received. It is intended that user code to service the request be provided within this function. For instance, if a file open indication is received, and the virtual machine is to take care of the details of servicing this indication, a call to the appropriate virtual response function can be made (**mv_fopen_resp**). If the user desires to take action bypassing the capabilities of the virtual machine, a call to the appropriate primitive response function (**mp_fopen_resp**) can be made after completing the actions needed to service the indication. If there is an error in servicing the request, a negative error response (**mp_err_resp**) can be made. This approach allows determining whether to use the virtual machine. It also allows a great deal of flexibility in how indications are handled.

The user indication functions take the form**:**

```
ST_VOID u_xxxx_ind (MMSREQ_IND *ind);
```

Parameters:

"xxxx" See **Volume 1 — Appendix D** for a list of all function names. Examples are:

fopen for a file open indication, or *read* for a read variable indication

ind This pointer points to the indication control data structure of type **MMSREQ_IND** that contain data specific to the indication received.

The determination of which user indication function to call is made using the function pointer array **mms_ind_serve_fun**. This is described on page 1-64. MMS-EASE defaults this function pointer array to the pre-named functions described in this manual. Normally, the user should not have to deal with this table; it is recommended that it not be modified.

Paired Primitive User Confirmation Function Class

Paired primitive user confirmation functions are called by MMS-EASE when a confirmation to a primitive request function (**mp_xxxx**) is received from the responding node. Any code desired can be placed into this pre-named function. This function is called so that the actions required by the application can be taken when confirmations to requests are received.

The primitive user confirmation functions take the form:

```
ST_VOID u_mp_xxxx_conf (MMSREQ_PEND *req);
```

Parameters:

"xxxx" See **Volume 1 — Appendix D** for a list of all function names. Examples are:

fopen for a file open confirmation, or *read* for a read variable confirmation

req — This pointer is the same pointer as the one returned from the primitive request function used to issue the request.

These functions can be used to perform any actions required by the application when a confirmation is received as a result of a primitive request function. If no error occurs (**req->resp_err = CNF_RESP_OK**), information from the PDU is available in the operation-specific data structure pointed to by the **req->resp_info_ptr** pointer. If an error occurred, **resp_err** will contain the error value. In this case, the operation-specific data structure pointed to by the **resp_info_ptr** may contain error information.

The determination of which specific primitive user confirm function to call (when the confirmation to a given type of service request is received) is made using the **mms_conf_serve_fun** function pointer array. This is described on page 1-64. MMS-EASE defaults this function pointer array to the pre-named functions corresponding to the functions available for confirms. Normally, the user should not have to deal with this table; it is recommended that it not be modified.

Error Response Function Class

Error response functions allow sending error responses to indications received from another node on the network. These functions are intended to be called from within a user indication function, after the requested action is determined to be invalid. However, a primitive error response function may be called from anywhere within your application program.

In general, when a primitive error response function is called, the following actions take place:

1. A message (an error response PDU) is constructed (encoded) using the information provided to the function call.
2. The appropriate command function is called to send the response message to the LLP.
3. Control is then returned to the calling program with a return value indicating success or failure of the function.

When control is returned to the calling program, it does not necessarily mean that the message has been delivered to the lower layers of software. If an error is detected later, MMS-EASE calls **u_llp_error_ind** function. This indicates that an error condition has occurred. Channel state information, contained in **mms_chan_info[chan].ctxt.xmit_pend** described on page 1-78, can be used to determine when the message has been delivered to the lower layers of the network.

Virtual Machine Request Function Class

Virtual machine request functions are used to issue requests through the virtual machine so that any necessary actions taken locally are automatically performed by the virtual machine. Generally, virtual request functions are available for most file and variable access operations. They are also provided to enhance the services of an existing primitive function such as reading multiple file segments using the **mv_fread** function versus one file segment using the **mp_fread** function.

The virtual request functions take the form:

```
MMSREQ_PEND *mv_xxxx(ST_INT chan, parm1,...,parmN);
```

Parameters:

"xxxx" — See **Volume 1 — Appendix D** for a list of all function names. Examples are: *fopen* for a file open request, or *read* for a read variable request.

chan — This is the channel number over which the request is to be made.

parm1...parmN — These are service specific parameters passed to the function during calling.

The virtual request function formats are tailored to the specific type of service being requested. For file operations, a file name is passed; for variable operations, the variable data is passed. This relieves the application program from the necessity of having to allocate and fill out any Operation-Specific data structures. The virtual machine automatically handles it.

Also, some virtual request functions will automatically group a series of requests together to perform the desired actions. For instance, even though the MMS specification allows only one file segment to be read per PDU, if the virtual request function is used to read a file, MMS-EASE will automatically combine several request PDUs and gather the responses. This makes the request look like a single transaction for multiple file segments.

Virtual Machine Response Function Class

Virtual Machine response functions can be used to respond to requests from another node. The virtual response functions automatically can take the actions required by the indication. If you do not want the virtual machine to take the required actions, a primitive response function should be called instead. Because these functions can automatically perform the desired actions, it is advisable to call these functions from within the user indication function called by MMS-EASE. However, this is not absolutely necessary as far as MMS-EASE is concerned. A virtual response function may be called from anywhere within your application program. The user indication function usually is the most logical place to issue responses.

The virtual response functions take the form:

```
ST_RET mv_xxxx_resp (MMSREQ_IND *ind, parm1,…,parmN);
```

Parameters:

"xxxx"	See **Volume 1 — Appendix D** for a list of all function names. Examples are: *fopen* for a file open response, or *read* for a variable read response.
ind	This pointer to a data structure of type **`MMSREQ_IND`** is passed to the user indication function containing data specific to the type of indication received.
parm1...parmN	These are service specific parameters passed to the function during calling.

Virtual Machine User Confirmation Function Class

Virtual Machine user confirmation functions are pre-named, user-defined functions that are called by MMS-EASE when the confirmation to a virtual request function is received. MMS-EASE does not require that your application program take any action within these functions, except for possibly releasing any allocated data structures used by the virtual machine using the **ms_clr_mvreq** function.

The virtual user confirmation functions take the form:

```
ST_VOID u_mv_xxxx_conf (MMSREQ_PEND *req);
```

Parameters:

"xxxx" See **Volume 1 — Appendix D** for a list of all function names. Examples are:

fopen for a file open confirmation, or *read* for a variable read confirmation.

req This pointer is the same pointer as the one returned from the primitive request function used to issue the request.

NOTE: When virtual request functions (**mv_xxxx**) are called, the input structures are automatically generated for their corresponding primitive request functions (**mp_xxxx**). The primitive functions do not free these structures (since they are provided by the user). The virtual machine functions do not free them, in case that information needs to be used in the user-defined **u_mv_xxxx_conf** functions. The **ms_clr_mvreq** support function will need to be called by the **u_mv_xxxx_conf** function to free any structures allocated by the virtual machine.

LLP Request Function Class

The LLP request function class consists of functions that the user calls to perform actions necessary to interface with the Lower Layer Provider (LLP). However, these are not directly supported by the MMS specifications. This includes registering process titles with the local OSI stack, de-registering them, etc. Typically, these functions are used to manipulate the network addressing elements required to communicate over the particular network used by MMS-EASE. By using the LLP request and user LLP function classes, the application program does not need to interact directly with the OSI stack to properly use the OSI services.

In general, the LLP request functions take the form:

ST_RET mllp_xxxx(parm1,...,parmN);

Parameters:

"xxxx" such as *reg_ar_name* for registering process

parm1...parmN These are parameters specific to the particular LLP request function being called.

The format of a LLP request function usually is dependent on the operation that is to be performed.

In general, when an LLP request function is called, the following actions are performed:

1. A call is made to the appropriate LLP command function.
2. Control is then returned to the calling program with a return code indicating the success or failure of the operation. If failure, an error code may be returned.

User LLP Function Class

The user LLP functions are pre-named, user-defined functions called by MMS-EASE when confirmations and indications of the low-level LLP specific association primitives are received. These are not supported directly by the MMS specification. These functions are used to detect association indications and confirmations that occur below the MMS initiate level. Because these functions are completely user-defined, anything can be placed in them that is required by your particular application.

In general, user LLP functions take the form:

```
extern ST_RET (*u_mllp_y_xxxx)(parm1,...,parmN);
```

Parameters:

"y" *a* for ACSE networks or (blank) for functions that are network independent.

"xxxx" *assoc_ind* when an association indication is received, or *assoc_conf* when an association confirmation is received.

parm1...parmN These are parameters specific to the particular user LLP function being called.

Support Function Class

Support functions are provided for handling operations specific to MMS-EASE. They are typically used for such tasks used to initialize or reset the MMS-EASE system, manipulate the variable definition tables of the virtual machine. In general, support functions take the form:

```
ms_xxxx(parm1,...,parmN);
```

Parameters:

"xxxx" = such as *add_named_type* for adding named types.

parm1...parmN = These are parameters specific to the particular support function being called.

Address Resolution Function Class

Functions of this class are user-defined functions called by MMS-EASE for resolution of variable addresses. Some of these functions are pointers to functions that you provide to do special processing during variable accessing. Some are pre-named functions called by MMS-EASE whenever a variable access is made by the virtual machine. These functions are only used for variable access. Please refer to the section on the address resolution functions in **Volume 2 — Module 5 — Variable Access** for more detail.

Interface Function Class

Interface functions provide the following functionality for MMS-EASE:

1. They provide the means for passing received indications to the user’s application.
2. They provide the means for passing received confirmations to the user’s application.
3. They provide all the LLP queue management functions needed internally by MMS-EASE.
4. They are used in conjunction with the MMS-EASE event notification mechanism to build fully asynchronous, event-based applications.

NOTE: Please refer to the individual product's release notes and installation guide to determine how MMS-EASE applications are notified of events for a specific operating system.

When an event occurs, such as an incoming message is detected, MMS-EASE informs the user application using the event mechanism supported for the specific platform. This mechanism merely notifies the user application of the occurrence of an event. The specific event has not yet been identified. After an event is reported, the user application must later call one of the other interface functions to have that event passed to their program in the form of an indication or confirmation.

Typically, only the interface function **ms_comm_serve** needs to be called to perform all the necessary functions. **ms_comm_serve** calls the other needed interface functions. However, access to the other interface functions is provided so the user can determine if, and when, messages should be sent, received, or responded to, under program control. This added control provided by MMS-EASE might be useful for applications that run in a time-critical environment where you may not want to service incoming indications or confirmations because you do not wish to interrupt your current processing. See page 1-88 for more information on this function.

Function Class Usage

The following figure depicts how the various function classes are used in a dialog between two application processes at least one of which uses MMS-EASE.

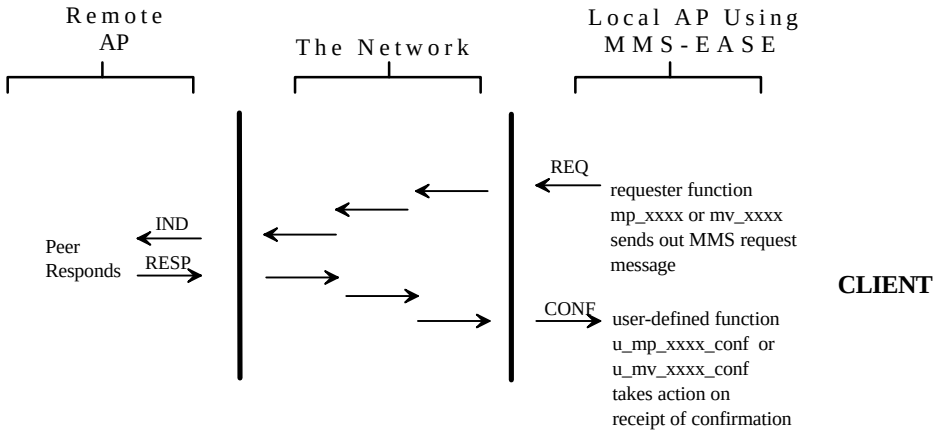


Figure 24: Function Class Usage in a MMS Dialog (Client)

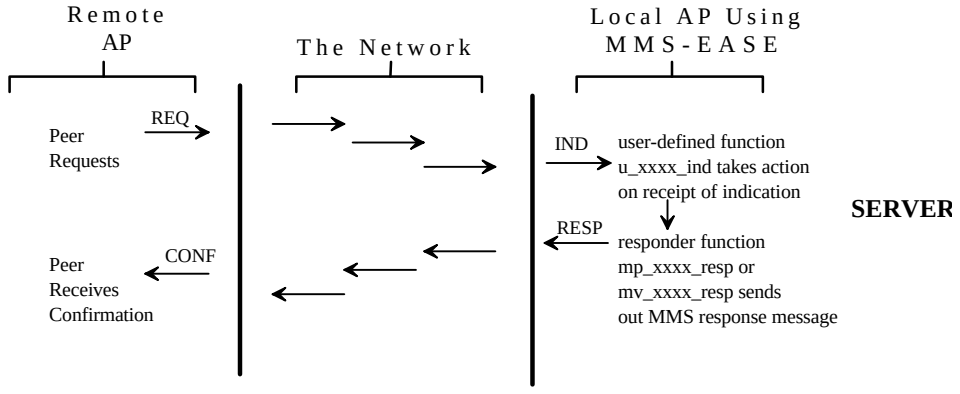


Figure 25: Function Class Usage in a MMS Dialog (Server)

User Defined Functions

All pre-named, user-defined indication and confirmation functions are referenced by an array of function pointers initialized to point to the **u_xxxx_ind**, **u_mv_xxxx_conf**, and **u_mp_xxxx_conf** functions defined in this manual. However, by manipulating the function pointer array, functions with user-supplied names can be called when a particular indication or confirmation is received. Because MMS-EASE defaults to the user defined functions described in this manual, these function pointer arrays do not need to be dealt with unless required by your application program.

```
extern ST_VOID (*mms_ind_serve_fun  [MAX_IND_SFUN+1]) (MMSREQ_IND *ind);
```

This function pointer array points to the functions to call when an indication is received. The array is based on the opcode of the specific MMS operation itself. This array is initialized to point to the **u_xxxx_ind** functions previously described in this manual. If you want a different function to be called when the next indication of a particular type is received, a pointer to that function can be simply written into the appropriate member of this array. MMS-EASE will then call that function each time the specified indication is received.

```
extern ST_VOID (*mms_conf_serve_fun [MAX_CONF_SFUN+1]) (MMSREQ_PEND *req);
```

This function pointer array points to the functions to call when a confirmation is received. The array is based on the opcode of the specific MMS operation itself. This array is initialized to point to the **u_mv_xxxx_conf** or **u_mp_xxxx_conf** functions previously described in this manual. If you want to have a different function called when the next confirmation of a particular type is received, a pointer to that function can be simply written into the appropriate member of this array. MMS-EASE will then call that function each time the specified confirmation is received.

Manipulation Of The User-Defined Function Pointers

To change which function is called when a particular indication or confirmation is received, please observe the following steps:

1. Write the function to be called.
2. Find the opcode of the MMS operation corresponding to the specific indication or confirmation with which you are concerned. See the table in **Volume 1 — Appendix B** to determine the specific opcode.

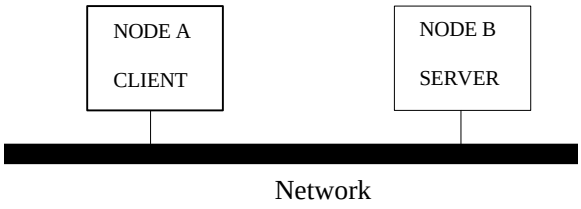
Write a pointer to your function in either the **mms_ind_serve_fun** or **mms_conf_serve_fun** array, using the opcode determined in step #2 as the index into the array.

1. General Sequence of Events

So far in the manual, the following topics have been covered:

- a brief summary of MMS and how it applies to factory floor devices,
- MMS addressing issues,
- MMS-EASE application interface, and
- a brief description of various function classes.

Now, it is necessary to outline the sequence of events that occur within the MMS-EASE environment. For the purposes of the following scenario descriptions, the assumption made is that the network consists of the following two nodes:



In this example, both nodes are using MMS-EASE to communicate with each other. Only the programmatic sequences are described here. Details such as setting up the operating system, creating the network directories, etc., are not covered in this section. Please refer to other sections of this manual, or to the documentation supplied with your network for more information.

Confirmed Services

The following is the sequence that occurs during a confirmed service. This sequence assumes that an association has already been established. This step is required before any other communications can take place between nodes A and B. See page 1-152 for an explanation of how an association is established. Now that an association is active, normal communications can take place. This involves Node A sending a request, Node B receiving an indication, Node B sending a response, and Node A receiving a confirmation.

The following scenario describes the sequence of events that occurs during this request-indication-response-confirmation sequence for the Identify service in more detail.

Please refer to **Volume 2 — Module 4 — VMD Support — Identify Service** for specific explanations of the functions referred to below:

Identify Request (Node A)

1. The MMS-EASE application sends an Identify request by calling **mp_ident**.
2. MMS-EASE then formats an Identify PDU and allocates and fills out a data structure of type **MMSREQ_PEND**. It sends the request to the OSI Stack and increments the transmit pending counter (**mms_chan_info[chan].ctxt.xmit_pend**). MMS-EASE returns to the calling program with a pointer to the pending request control structure. When the OSI Stack receives the Identify PDU, it notifies MMS-EASE. During a call to **ms_comm_serve**, MMS-EASE detects this event and decrements the transmit pending counter (See Note #1).

Identify Indication (Node B)

3. The OSI Stack notifies MMS-EASE that an event has occurred.
4. This indicates to the application that it should call **ms_comm_serve** to process pending events (See Note #2). The application calls **ms_comm_serve** in a loop while **ms_comm_serve** returns **SD_TRUE**. The return value of **SD_TRUE** indicates that there are more pending events.

5. Within the **ms_comm_serve** function, an indication is detected. The PDU is parsed and an indication control data structure of type **MMSREQ_IND** is allocated and filled in. If an error parsing the PDU is detected, MMS-EASE automatically sends a reject and makes a call to **u_reject_ind** indicating to the user that a reject was sent.
6. When the application calls **ms_comm_serve**, MMS-EASE services the next indication. Note that indications are serviced in "First In First Out" order and that at most one callback function is invoked during a call to **ms_comm_serve**.
7. From within **ms_comm_serve** MMS-EASE calls the user indication function (**u_ident_ind**).

Identify Response (Node B)

8. Within the user-defined **u_ident_ind** function, you call either the **mp_ident_resp, mv_ident_resp** or **mp_err_resp** function to service the indication. Calling a response function from within the **u_ident_ind** function is not strictly necessary; you may choose to do so at any time.
   Once the response function returns, the **MMSREQ_IND** structure is freed.

Identify Confirm (Node A)

9. The OSI Stack notifies MMS-EASE that an event has occurred.
10. This indicates to the application that it should call **ms_comm_serve** to process pending events.
11. The application calls **ms_comm_serve** in a loop while **ms_comm_serve** returns **SD_TRUE.** The return value of **SD_TRUE** indicates that there are more pending events. Within the **ms_comm_serve** function, a confirmation is detected. The confirmation PDU is then parsed and paired with the **MMSREQ_PEND** structure used to originate the request. If an error parsing the PDU is detected, MMS-EASE automatically sends a reject back and makes a call to **u_reject_ind** indicating to the user that a reject was sent.
12. From within **ms_comm_serve,** MMS-EASE calls the user-defined **u_mp_ident_conf** function. Within this function, you can examine the **MMSREQ_PEND** structure to determine if the request succeeded or failed. You also can examine the Operation-Specific data structure (**ident_resp_info**) pointed to by **resp_info_ptr** to evaluate the additional information sent in the confirmation.
13. Upon returning from the **u_mp_ident_conf** function, the **MMSREQ_PEND** structure is freed

NOTES:

1. The OSI Stack informs MMS-EASE when it has received and accepted a request from the MMS-EASE application. This, however, does not guarantee that the request will be sent on the network, only that the Stack will make an attempt to send the request. If the attempt fails, the Stack will issue an event to MMS-EASE indicating the failure. MMS-EASE, in turn, will notify the application by calling the user callback confirm function and returning an error code in the **MMSREQ_PEND** structure.
2. MMS-EASE reports the occurrence of events to user applications using some means available in the operating system, such as signals under UNIX or Event Objects under Win32. The techniques used to implement event notifications in MMS-EASE vary greatly depending on the features available in the native operating system. Although similar in nature, not all MMS-EASE products support the same event notification method. Please refer to the individual product's release notes and installation guide to determine how events are handled for a specific operating system. Note that it is not strictly required that you write an event-based application. Instead, you may choose to poll for events by periodically calling **ms_comm_serve**.

Unconfirmed Services

The following is the sequence that occurs during an unconfirmed service. See page 1-152 for an explanation of how an association is established. Now that an association is active, normal communications can take place. Outwardly this involves Node A sending a request, Node B receiving an indication. The following scenario describes the sequence of events that occurs during this request-indication sequence for the UnsolicitedStatus service in more detail.

Please refer to **Volume 2 — Module 4 — VMD Support — UnsolicitedStatus Service** for specific explanations of the functions referred to below:

UnsolicitedStatus Request (Node A)

1. Node A sends an UnsolicitedStatus request by calling **mp_ustatus**.
2. MMS-EASE then formats an UnsolicitedStatus PDU, allocates and fills out a data structure of type **MMSREQ_PEND**. It sends the request to the OSI Stack and increments the transmit pending counter (**mms_chan_info[chan].ctxt.xmit_pend**). MMS-EASE returns to the calling program with a pointer to the pending request control structure. When the OSI Stack receives the UnsolicitedStatus PDU, it notifies MMS-EASE. During a call to **ms_comm_serve**, MMS-EASE detects this event and decrements the transmit pending counter (See Note #1).

UnsolicitedStatus Indication (Node B)

3. The OSI Stack notifies MMS-EASE that an event has occurred.
4. This indicates to the application that it should call **ms_comm_serve** to process pending events (See Note #2). The application calls **ms_comm_serve** in a loop while **ms_comm_serve** returns **SD_TRUE.** The return value of **SD_TRUE** indicates that there are more pending events.
5. Within the **ms_comm_serve** function, an indication is detected. The PDU is parsed and an indication control data structure of type **MMSREQ_IND** is filled in. If an error parsing the PDU is detected, MMS-EASE automatically sends a reject and makes a call to **u_reject_ind** indicating to the user that a reject was sent.
6. When the application calls **ms_comm_serve**, MMS-EASE services the next indication. Note that indications are serviced in “First In First Out” order and that at most one callback function is invoked during a call to **ms_comm_serve**.
7. From within **ms_comm_serve**, MMS-EASE calls the user indication function (**u_ustatus_ind**).
8. Once **u_ustatus_ind** completes, the **MMSREQ_IND** structure is freed.

VMI vs. PPI Regarding Sequence of Events

Generally, the same sequence of event occurs for both the PPI and the VMI. The main difference is that the VMI (when available) performs additional processing to make MMS-EASE easier to use. The VMI may require more configuration, but it generally results in much simpler and shorter code.

Both the PPI and VMI have similar requirements. For example, a given MMS service must be enabled in the **mmsop_en.h** file before the corresponding MMS-EASE functions (either VMI or PPI) will work. The appropriate interface functions (usually **ms_comm_serve**) must be called before a user indication or confirm function will be called.

The same features described in the general sequence of events are available to the user application regardless of whether it is making use of the PPI or VMI. For example, the **cmd_service** structure may be used by the user application in both cases.

2. General Data Structure Classes

Besides the function classes described earlier, there is another important element of the MMS-EASE interface: the data structures. These structures are used to hold information passed back and forth between MMS-EASE and the user code. The available types of data structure classes are:

1. **Operation-Specific Data Structures** are specific to a particular MMS service. They are explained individually in each module.
2. **Request and Indication Control Data Structures** relate to general request and indication control information for requests, responses, indications and confirmations. See page 1-70 for more information on these structures.
3. **MMS-EASE System Data Structures** contain presentation context and error information. Also included in data structures of this type are configuration variables necessary to configure the MMS-EASE environment. See page 1-146 for more information on these structures.
4. **Virtual Machine Data Structures** are used to support the virtual machine interface and contain the various domain, context management, variable, and type databases. They are explained individually in each module.

Operation-Specific Data Structures

There is usually at least one Operation-Specific data structure per MMS service. In some cases, there are two data structures: one for requests or indications, and one for responses or confirmations. The format for each structure is specific to the type of service involved. For instance, the **mp_fopen** function uses a structure of type **FOPEN_REQ_INFO** containing information applicable to the File Open service request only. An identical structure is used when an indication is passed to the **u_fopen_ind** function. Likewise, the **mp_fopen_resp** function uses a structure of type **FOPEN_RESP_INFO** that is also passed to the **u_mp_fopen_conf** function.

Before calling a primitive request function, an appropriate Operation-Specific Data Structure should be created to contain the data for that request. If using the virtual machine to generate requests, the data required by the virtual request function only needs to be passed. The virtual machine automatically creates any necessary Operation-Specific data structures.

When an indication is received, a pointer is provided to an Operation-Specific data structure containing the information for that request. That data may then be accessed, and processed as desired. If the response also requires an Operation-Specific data structure, one must be created and filled in before the response function is called (only when the paired-primitive interface is used). Again, if the response is generated using a virtual response function, the virtual machine creates the necessary Operation-Specific data structures.

When using the primitive functions, any allocated Operation-Specific data structures can be freed after the request is issued. For structures allocated by MMS-EASE as a part of a virtual machine request, they must be freed by calling the support function **ms_clr_mvreq** when your application program and MMS-EASE are done with the information contained in those structures. This is typically done from within the **u_mv_xxxx_conf** function.

SPECIAL NOTE ON SOME OPERATION-SPECIFIC DATA STRUCTURES

ALLOCATION EXCEPTIONS:

Please note that some Operation-Specific Data Structures have some members that are ”commented out.“ This is because the size of these members cannot be determined ahead of time. In these cases, follow the allocation procedures shown for those structures to allocate enough memory to hold the members that have been commented out. These special allocation procedures are described in notes that have been placed at the end of the Operation-Specific Data Structure definition. See **Volume II —Module 5 — DELVAR_REQ_INFO** for an example. To access those members of Operation-Specific Data Structures that have been commented out, your pointers will need to be manipulated manually (by incrementing the base pointer) to point to those ”commented out members“ of these structures.

Indication And Request Control Data Structures

These data structures are provided by MMS-EASE for tracking each outstanding service request and indication. There are two types of control data structures:

pending request control data structure (**MMSREQ_PEND**)
indication control data structure (**MMSREQ_IND**)

When a request is issued, by using a MMS-EASE **mp_xxxx** or **mv_xxxx** function, MMS-EASE provides a pending request control data structure for tracking the outstanding request. A pointer to this structure is returned to the MMS-EASE user application. This control structure is valid until a confirm for the request is received by MMS-EASE, and the **u_mp_xxxx_conf** or **u_mv_xxxx_conf** function is invoked and returned from.

Similarly, when an indication is received, MMS-EASE parses the PDU, then allocates and fills in an indication control data structure. A call to the appropriate **u_xxxx_ind** function is made, and a pointer to the indication control data structure is passed to the function. MMS-EASE only calls the indication function when **ms_comm_serve** is called. The indication control structure remains active until the user program services the request by calling a **mp_xxxx_resp**, **mv_xxxx_resp** or **mp_xxxx_err** function.

Pending Request Control Data Structure

MMSREQ_PEND

The structure of this type is used to track requests and confirmations. These structures are allocated by MMS-EASE as requests are made. When the user application issues a request by calling a MMS-EASE **mp_xxxx** or **mv_xxxx** function, MMS-EASE provides a control structure for tracking the outstanding request. A pointer to this structure is returned to the MMS-EASE application. When a confirmation is received, MMS-EASE locates the pending request control structure used to initiate the request. At this point, MMS-EASE calls the appropriate **u_xxxx_conf** function and passes a pointer to the request control structure. When the user function returns, the request control structure is released.

```
typedef struct rqdat
  {
  DBL_LNK      link;            /* DO NOT USE */
  ST_INT       pri;
  ST_INT       chan;
  ST_INT        llp;
  ST_UINT       context;
  ST_INT       op;
  ST_UINT32    id;
  ST_RET       resp_err;
  ST_INT       cancl_state;
  time_t       req_time;
  ST_BOOLEAN   req_info_pres;
  ST_VOID       *req_info_ptr;
  struct       list_of_mods   mods;
  time_t       resp_time;
  ST_BOOLEAN   resp_info_pres;
  ST_VOID       *resp_info_ptr;
  ST_INT        rxbuf_type;
  ST_UCHAR     *rxbuf;
  ST_VOID       (*resp_rcvd_fun)(struct rqdat *req);
  struct        csi             cs;
  struct        cmd_service     s;
  ST_VOID       *usr;
  ST_EVENT_SEM done_sem;
  } MMSREQ_PEND;
```

Fields:

`link` /* FOR INTERNAL USE ONLY, DO NOT USE */

`pri` This contains the priority of the request. This is currently not used.

`chan` This contains the channel number over which the request was sent and over which the confirmation should be received.

`llp` This indicates what type of Lower Layer Provider (LLP) the request was sent over. The available option at this time is **`ACSE30_LLP`** for ACSE V3.0.

`context` This contains the presentation context used to send the request. This variable should not be changed. See **`P_context_sel`** on page 1-146 for more information.

`op` This contains the MMS opcode of the request. Normally you do not need to deal with opcodes, but this is provided for your information. See **Volume I — Appendix B** for a list of valid opcodes.

`id` This contains the InvokeID of the request.

`resp_err` This contains an error code indicating whether a correct confirmation was received. It is only valid after the confirm is received and is set to zero when a request is first made. It can take on the following values shown represented by defined constants and their corresponding hexadecimal values:

`CNF_RESP_OK` (0x6600)	Valid response received.
`CNF_PARSE_ERR` (0x6601)	Confirm PDU Parse error found locally.
`CNF_REJ_ERR` (0x6602)	Request rejected by peer.
`CNF_ERR_OK` (0x6603)	Error response received.
`CNF_DISCONNECTED` (0x6604)	Association disconnected or aborted.
`CNF_CHAN_OP_ERR` (0x6605)	Wrong operation (opcode) for invoke id.
`CNF_CANST_ERR` (0x6606)	Cancel state error.
`CNF_ASS_REQ_REJECTED` (0x6607)	Associate request rejected.
`CNF_ASS_RESP_PARAM` (0x6608)	Invalid Associate Response parameters.
`CNF_ASS_USER_REJ_CONF` (0x6609)	Initiate confirmation rejected locally by user.
`CNF_REM_FOPEN` (0x660A)	FileOpen rejected remotely.

CNF_INIT_PARAM (0x660B)	Initiate parameters rejected.
CNF_REM_FREAD (0x660C)	FileRead rejected remotely.
CNF_LOC_FWRITE (0x660D)	FileWrite rejected locally.
CNF_REM_FCLOSE (0x660E)	FileClose rejected remotely.
CNF_LOC_FCLOSE (0x660F)	FileClose rejected locally.
CNF_MVREAD_RESP_PARAM (0x6610)	Invalid virtual machine Read response parameters.
CNF_VM_RESP_ERR (0x6611)	Could not send response.
CNF_LLC_SEND_ERROR (0x6612)	Error sending request rejected at LLC.
CNF_REQ_NOT_DONE (0x6613)	The confirm has not yet been received and there is no other activity (such as a disconnect). This is the initial value returned immediately after the request is sent.

`cancl_state` This indicates the current state of a pending cancel operation:

CANCEL_REJECTED	Cancel attempted; reject occurred — unknown state.
CANCEL_FAILED	Cancel attempted but failed; back in no-cancel state.
NO_CANCEL	No Cancel has been attempted.
CANCEL_REQUESTED	Cancel request sent, no response received.
POS_CANCEL_RCVD	Cancel confirm (+) received, but service confirm(-) not received yet.
POS_CANCEL_DUE	Service confirm (-) due to cancel received, but no cancel confirm (+) received.
NEG_CANCEL_DUE	Service confirm (+) or (-) received due to some other reason than cancel, but cancel confirm (-) not received.
CANCELLED	Both service confirm (-) and cancel confirm (+) received, waiting for **u_mp_xxxx_conf** or **u_mv_xxxx_conf** to be called.

`req_time` This is the time (in **time_t** format) that the request was sent.

`req_info_pres` **SD_FALSE**. There is no additional information contained in the PDU. There were no request parameters.

SD_TRUE. There is additional information contained in the PDU pointed to by **req_info_ptr**.

`req_info_ptr` This points to any request information sent out in the request. It typically points to the Operation-Specific Data Structure passed to the request function.

`mods` This is a structure of type **LIST_OF_MODS** containing information about the modifier that might have been sent out in the request. This is strictly to be used by the application so that a pointer to the original request information will be present in a user confirm function. The modifier can be examined at this point. See **Volume 3 — Module 7 — Modifier Handling** for a detailed description of this structure.

`resp_time` This contains the time (in **time_t** format) at which the confirmation was received.

`resp_info_pres` **SD_FALSE**. There is no additional information contained in the PDU. There were no response parameters.

SD_TRUE. There is additional information contained in the PDU. This information is stored in the Operation-Specific structure pointed to by **resp_info_ptr**.

`resp_info_ptr` This points to any response information contained in the confirmation PDU. Typically, such information is stored in the Operation-Specific Data Structure. This pointer is not valid until the confirmation is received.

```
rxbuf_type      /*  FOR INTERNAL USE ONLY, DO NOT USE  */
rxbuf
```

`cs` This is a structure of type **`CSI`** containing received companion standard information. See page 1-61 for more information on this structure.

`s` This is a structure of type **`CMD_SERVICE`** containing additional user-specified outstanding information relating to this request. See page 1-64 for more information on this structure

`usr` This is a pointer to some user specific information that can be associated with this request. The contents and meaning of the data pointed to by **`usr`** is completely user-defined. MMS-EASE does not use this pointer.

`done_sem` This field is related to the multi-threaded functionality of MMS-EASE. If this field is not null, MMS-EASE will set the event semaphore when the request is completed and will not call the **`u_mp_xxxx_conf`** or **`u_mv_xxxx_conf`** function.

Warning: Elements marked as ”DO NOT USE“ in the **`MMSREQ_PEND`** structure are for internal MMS-EASE USE only. They should **not** be modified or accessed in any manner by your application program.

Indication Control Data Structure

`MMSREQ_IND`

The structure of this type is used to track indications and responses. It corresponds to the ”transaction object“ of the MMS specification. These structures are allocated by MMS-EASE as requests from remote applications are received.

```
typedef struct rspdat
  {
  DBL_LNK     link; /* DO NOT USE       */
  ST_INT      pri;
  ST_INT      chan;
  ST_INT      llp;
  ST_UINT     context;
  ST_INT      op;
  ST_UINT32  id;
  ST_BOOLEAN req_info_pres;
  ST_VOID     *req_info_ptr;
  struct      list_of_modsmods;
  ST_INT      cancl_state;
  time_t      ind_time;
  ST_INT      rxbuf_type;
  ST_UCHAR   *rxbuf;
  struct      cmd_service s;
  ST_BOOLEAN add_addr_info_pres;
  struct      llp_addr_info     *add_addr_info;
  struct      csi               cs;
  ST_VOID     *usr;
  } MMSREQ_IND;
```

Fields:

`link` /* FOR INTERNAL USE ONLY, DO NOT USE */

`pri` This contains the priority of the indication. This is not currently used.

`chan` This contains the channel number over which the indication was received.

`llp` This indicates what type of Lower Layer Provider (LLP) from which the indication was received. The available option at this time is **`ACSE30_LLP`** for ACSE V3.0.

`context` This contains the presentation context mask corresponding to this indication. This variable should not be changed. See page 1-146 for more information.

`op` This contains the MMS opcode of the particular service request received. MMS-EASE automatically decodes incoming messages and calls the appropriate function. Normally, your application programs will not have to deal with this variable.

`id` This contains the InvokeID of the indication.

`req_info_pres` **`SD_FALSE`**. There is no additional information contained in the PDU.

`SD_TRUE`. This information is stored in the Operation-Specific structure pointed to by **`req_info_ptr`**.

`req_info_ptr` This points to the additional information contained in the PDU. Typically, such information is stored in the Operation-Specific data structure.

`mods` This is a structure of type **`LIST_OF_MODS`** indicating that a modifier may be present. Do NOT alter any values present since unpredictable results may occur. See **Volume 3 — Module 7 — Modifier Handling** for a detailed description of this structure.

`cancl_state` This indicates the current state of a pending cancel operation for the indication. Available options are:

`CANCEL_FAILED`	Cancel attempt denied; back in no-cancel state
`NO_CANCEL`	No cancel has been attempted
`CANCELING`	Cancel Indication received, no response sent
`POS_CANCEL_SENT`	Cancel Response (+) sent, but no service response (-) sent yet

Note: No other states are needed at the responder since the service response or error is never sent first by MMS-EASE.

`ind_time` This is the time (in **`t_time`** format) that the indication was received.

`rxbuf_type` /* FOR INTERNAL USE ONLY, DO NOT USE */
`rxbuf`

`s` This is a structure of type **`CMD_SERVICE`** containing the command service information related to this indication. This information is to be implemented by the user's application program. See page 1-75 for a detailed description of this structure.

`add_addr_info_pres` **`SD_FALSE`**. There is no specific LLC addressing information contained in the PDU.

`SD_TRUE`. There is specific LLC addressing information contained in the PDU pointed to by **`add_addr_info`.**

`add_addr_info` This is a structure of type **`LLP_ADDR_INFO`** pointing to the specific LLC addressing information. The flag **`add_addr_info_pres`** indicates whether there is data present.

`cs` This is a structure of type **`CSI`** containing companion standard information. See the next page for more information on this structure.

`usr` This is a pointer to some user specific information that can be associated with this indication. The contents and meaning of the data pointed to by **`usr`** is completely user-defined. MMS-EASE does not use this pointer.

Warning: Variables marked as "DO NOT USE" in the **`MMSREQ_IND`** structure are for internal MMS-EASE use. They should not be modified or accessed in any manner by your application program.

Companion Standard Data Structure

IS and DIS differ in the handling of companion standard information. DIS allows sending of companion standard information only for selected services. IS allows sending companion standard information on any Request or Response PDU. All companion standards are written for the IS version of MMS only. The following structure is a member of both the **MMSREQ_IND** and **MMSREQ_PEND** structures. It is used to pass companion standard information between MMS-EASE and the user application. This structure will be used as a global input parameter to all MMS requests and responses sent by the user application. The flag, **cs_pres**, must be set to **SD_TRUE** to receive companion standard information. This flag is reset after being used.

```
struct csi
  {
  ST_BOOLEAN cs_pres;
  ST_INT      cs_len;
  ST_UCHAR    *cs_ptr;
  };
typedef struct csi  CSI ;

extern  struct csi cs_send;
```

Fields:

cs_pres **SD_FALSE**. There is no companion standard information contained in the PDU.

SD_TRUE. There is companion standard information contained in the PDU pointed to by **cs_ptr**.

cs_len This indicates the length of the ASN.1 data element containing the companion standard information pointed to by **cs_ptr**.

cs_ptr This is a pointer to the ASN.1 data element containing the companion standard information.

Command Service Data Structure

The following structure is a member of both the **MMSREQ_IND** or **MMSREQ_PEND** structures. It contains user specific information regarding the service of the indication or request/confirmation. The information in this structure is strictly for the user's program. It does not need to be modified or read by your program unless desired.

```
struct cmd_service
  {
  ST_RET     (*serve_fun)(ST_VOID *p);
  ST_CHAR    *cmd_info;
  ST_INT32   cmd_stat;
  ST_INT     aux_flags[6];
  };
typedef struct cmd_service CMD_SERVICE;
```

serve_fun() **(Indications)**

The function pointed to by this pointer is called every time that **ms_comm_serve** is called, if the indication is still active. By default, MMS-EASE sets this pointer to point to a null function when the indication is first put on the active list. However, within the **u_xxxx_ind** function, the pointer can be set to point to another function. Then, MMS-EASE will call that function every time **ms_comm_serve** is called while the indication is active. A pointer to the corresponding **MMSREQ_IND** structure is passed to the **serve_fun** function. Currently, the return value is ignored.

serve_fun() **(Confirmations)**

The function pointed to by this pointer is first called when the confirmation PDU is parsed and paired with the **MMSREQ_PEND** structure. This occurs during a call to the **ms_comm_serve** function and before the **u_mp_xxxx_conf** or **u_mv_xxxx_conf** functions are called. This allows you to have MMS-EASE call a function when a specific confirmation is received, in addition to the standard **u_xxxx_conf** function. If you decide to use the **serve_fun** function, set the pointer upon returning from the **mp_xxxx** or **mv_xxxx** function but before the confirmation is received. By default, MMS-EASE sets this function pointer to a null function when the request is first sent. It is ignored if it is not being used. A pointer to **MMSREQ_PEND** is passed to the **serve_fun** function. Currently, the return value is ignored.

cmd_info **cmd_stat** **aux_flags**

These fields can contain any user-defined information. MMS-EASE ignores these fields.

cmd_info	This is a pointer to command specific information.
cmd_stat	This indicates a general command status.
aux_flags	This is an array of auxiliary status flags.

3. MMS-EASE Network Specific Issues

This section deals with network specific issues such as channel management and association control.

Communication Channels

Communication channels represent potential associations with other applications, and each active association takes up one channel. In general, OSI hardware and software have limited resources; it can only support a certain number of associations simultaneously. This translates into a maximum number of channels that a MMS-EASE implementation can support; some MMS-EASE implementations support 64, some 200, and some more. Assume that an OSI communications stack supports 64 active connections or contexts. Then an application would only be able to initiate (or accept) associations to at most 64 other applications at the same time. If communication with more than 64 other applications is needed, it would have to conclude some associations, and initiate new ones as necessary. If there were two applications running simultaneously on the same computer and both were sharing the same OSI stack, the contexts for that stack would have to be divided between them. The contexts in a stack represent a finite resource that must be shared among the total number of potential associations.

Each MMS-EASE user application must first register its AR Name on a channel before it can initiate or accept associations on that channel. Each channel is assigned a number between zero and the number of channels minus one. An application simply passes the appropriate channel number to the MMS-EASE register function to register that channel. All channels are identical in function, so it makes no difference how the channel numbers are assigned among the local applications. If some of the channels have not been registered to the application, then those channels are unavailable. Please refer to the documentation for your particular MMS-EASE implementation for details on the number of channels it supports.

Any AR Name may be registered on a given channel with the provision that it is contained in the **D**irectory **I**nformation Base or **DIB** as a local AR Name, not a remote one. See page 1-82 for more information on the DIB.

NOTE: See page 1-39 for more information on AR Names.

Channel Information Structure

The following array of structures, **mms_chan_info[]**, contains the negotiated context management information for each active channel, and the preferred initiate parameters for idle channels. The channel number is used as the index into this array.

```
struct mchaninfo
  {
  struct
    {
    ST_INT   llp_type;
    ST_ULONG chan_state;
    ST_INT   xmit_pend;
    ST_UINT  contexts;
    ST_CHAR  *locl_ar;
    ST_CHAR  *rem_ar;
    } ctxt;
  struct
    {
    struct   vmd_ctrl           *vmd;
    struct   domain_objs  *aa_objs;
    } objs;
  ST_INT16   version;
  ST_INT32   segsize;
  ST_INT16   maxpend_req;
  ST_INT16   maxpend_ind;
  ST_CHAR    max_nest;
  ST_UCHAR   param_supp[2];
  ST_UCHAR   service_req[11];
  ST_UCHAR   service_resp[11];
  ST_INT      file_blk_size;
  ST_INT      download_blk_size;
  ST_CHAR    *user_info;
  };
typedef struct mchaninfo MCHANINFO;
                         /* ptr to table of chan ctrl structs   */
                         /* [max_mms_chan], allocated at init   */
extern struct mchaninfo *mms_chan_info;
```

Fields:

`llp_type` This is the type of **L**ower **L**ayer **P**rovider (LLP). The only LLP supported at this time is **ACSE30_LLP** (ACSE V3.0).

`chan_state` This contains the current state of the channel that is represented by a set of bits. The various allowable channel states are described below. Some states can co-exist with other states. In this case, the state bits are ORed together.

DEFINED CONSTANT	DESCRIPTION
M_IDLE	Channel is idle. Association requests may be sent on this channel if there is an AR Name registered.
M_LISTEN	A channel is in listen mode with no active associations. Association requests can be received on this channel.
M_STOPPING_LISTEN	A stop listen has been requested for this channel, but the channel has not yet exited listen mode.
M_REPOST_LISTEN	A listen has been automatically reposted to this channel as a result of a reject or negative response to an associate request received on this channel, but the channel is not yet in listen mode again.

CHANNEL STATES (cont'd)

DEFINED CONSTANT	DESCRIPTION
M_ASS_REQ_PEND	An associate request has been sent on this channel, but the confirmation has not yet been received.
M_ASS_IND_RCVD	An associate indication has been received for this channel, but has not been responded.
M_ASS_RESP_PEND	A positive associate response has been sent to the OSI Stack, but it has not yet been accepted (waiting for associate response done).
M_NEG_ASS_RESP_PEND	A negative associate response has been sent to the OSI Stack, but it has not yet been accepted (waiting for associate response done).
M_ASSOCIATED	A positive associate response has been send to the OSI Stack, and the associate response done has been sent back confirming it has been accepted.
M_INIT_REQ_PEND	A MMS Initiate request has been sent on this channel, but the Initiate confirmation has not yet been received.
M_INIT_URESP_WAIT	A MMS Initiate indication has been passed to the user (using a call to **u_init_ind**), but the Initiate response has not been sent.
M_INIT_RESP_PEND	The Initiate response has been sent to the OSI Stack, but it has not yet been accepted.
M_ACTIVE	There is an association currently active for this channel.
M_CONCL_REQ_PEND	A MMS Conclude request has been sent on this channel but the confirmation has not yet been received.
M_CONCL_URESP_WAIT	A MMS Conclude indication has been received and passed to the user (using a call to **u_conclude_ind**), but it has not yet been responded to.
M_REL_IND_WAIT	The MMS Conclude response has been successfully delivered to the OSI Stack and is now waiting to receive the release indication.
M_REL_IND_RCVD	A release indication has been received, but not yet responded to. This channel state will not normally be displayed. It is used internally by MMS-EASE.
M_REL_REQ_PEND	A release request has been sent, but the confirmation has not yet been received.
M_REL_RESP_PEND	A release response has been sent to the OSI Stack, but has not yet been accepted — now waiting for release response done.
M_ABORT_REQ_PEND	An Abort request has been sent to the OSI Stack, but has not yet been accepted — waiting for abort done.
M_ABORT_IND_RCVD	An Abort indication has been received, but has not yet been passed to the user. This channel state will not normally be displayed. It is used internally by MMS-EASE.
M_WAIT_DISCONNECT	Waiting for a disconnect.
M_ACSE_REJECT_PEND	An Associate reject is pending.

mms_chan_info (cont'd)

xmit_pend — This element is incremented by MMS-EASE each time an attempt is made to transmit a message to the OSI Stack for this channel. It is decremented each time the OSI Stack accepts a message from this channel. It can be used to determine when messages have been delivered to the OSI Stack.

contexts — These are the presentation contexts negotiated for this channel. See **P_context_sel** for possible values.

ctxt.locl_ar — This is the local AR Name registered for this channel.

ctxt.rem_ar — This is the remote AR Name with which an association has been established. It is valid only when the channel is active.

vmd — This is a pointer to the VMD control structure of type **VMD_CTRL** corresponding to the particular VMD supported as a server on this channel. See **Volume 2 — Module 4 — VMD Control** for more information on multiple VMD support and the VMD control structure.

aa_objs — This is a pointer to the domain object structure of type **DOMAIN_OBJS** containing the list of application-association-specific objects defined for this channel. See **Volume 2 — Module 6 — Domain Management** for more information on the domain object structure.

version — This is the version of the MMS standard that was negotiated for this channel. This is equal to 0 for the DIS version of MMS or equal to 1 for the IS version of MMS.

PREFERRED INITIATE PARAMETER: If the channel is not active, the value of this member is used as a preferred initiate parameter. See page 1-149 for more information.

seg_size — This is the negotiated segment size. This is equivalent to the maximum number of bytes that can be sent in a MMS PDU over this channel.

PREFERRED INITIATE PARAMETER: If the channel is not active, the value of this member is used as a preferred initiate parameter.

maxpend_req — This is the maximum number of confirmed requests that the local node can have outstanding. A response from the remote node must be received before the local node can issue another confirmed request.

PREFERRED INITIATE PARAMETER: If the channel is not active, the value of this member is used as a preferred initiate parameter.

maxpend_ind — This is the maximum number of confirmed requests that the remote node can have outstanding. The local node must issue a response before another request can be received from the remote node.

PREFERRED INITIATE PARAMETER: If the channel is not active, the value of this member is used as a preferred initiate parameter.

max_nest — This is the maximum data structure nesting level negotiated for this channel.

PREFERRED INITIATE PARAMETER: If the channel is not active, the value of this member is used as a preferred initiate parameter.

param_supp — These are the parameter support options negotiated for this channel. See page 1-149 for more information on this parameter.

PREFERRED INITIATE PARAMETER: If the channel is not active, the value of this member is used as a preferred initiate parameter.

service_req — These are the vertical conformance building blocks supported as a requester on this channel. See page 1-149 for more information.

PREFERRED INITIATE PARAMETER: If the channel is not active, the value of this member is used as a preferred initiate parameter.

service_resp — These are the vertical conformance building blocks supported as a responder on this channel. See page 1-149 for more information.

`file_blk_size` This is the maximum number of bytes that can be sent or received in a given file segment of a FileRead PDU. This parameter is not negotiated. It is calculated by MMS-EASE using the negotiated **`seg_size`** parameter.

`download_blk_size` This is the maximum number of bytes that can be downloaded in a given segment of a DownloadSegment PDU. This parameter is not negotiated. It is calculated by MMS-EASE using the negotiated **`seg_size`** parameter.

`user_info` This is a pointer to some user specific information that can be associated with this channel. The contents and meaning of the data pointed to by **`user_info`** is completely user-defined. MMS-EASE does not use this pointer.

Channel Management Issues

For all MMS-EASE applications, the following functions need to be performed in the sequence shown:

1. **Activate** a local AR Name.Use the **`mllp_act_ar_name`** function for this operation.
2. **Register** an active local AR Name to a specific channel.
   Once the AR Name is activated, assign that AR Name to a specific channel using the **`mllp_reg_ar_name`** function.
3. **Post a listen** to a specific channel if the application needs to receive associate requests.
   Once the channel is registered, associate requests can now be issued on that channel. However, the channel will not receive associate requests until a listen is posted for that channel using the **`mllp_ass_listen`** function. Once a listen is posted, associate requests can no longer be sent. If you want to issue an associate request on a channel that is in the ”Listening“ mode, cancel the listen using the **`mllp_stop_ass_listen`** function for that channel.

Association Control Issues

There are several association control issues related to the association establishment mechanism and the manipulation of the addressing elements.

On an ACSE-network, MMS-EASE calls a function named **`u_mllp_a_assoc_ind`** and passes a pointer to a data structure of type **`ACSE_ASSINFO`**. When a positive associate confirmation is received, MMS-EASE calls the **`u_mllp_a_assoc_conf`** function and passes a pointer to a structure of type **`ACSE_ASSINFO.`**

Presentation Context Selection

A MMS Presentation Context represents a ”dialect of the MMS language.“ MMS is referred to as the ”core“ context. Additional MMS Presentation Contexts correspond to the MMS Companion Standards. These are based on the main MMS standard, but customized for robots, numerical controllers, programmable controllers, and process controls. Contexts can be added using the **`mllp_add_p_context`** function. See page 1-159 for more information. During the Initiate exchange, applications negotiate which Presentation Contexts they will use throughout the association.

Addressing Information With MMS-EASE

All addressing and ACSE information with MMS-EASE is provided using human readable character strings called AR Names. See page 39 for more information on AR Names. AR Names are used to activate local addresses and register channels and initiate associations with remote nodes.

The Directory Information Base (DIB)

The **D**irectory **I**nformation **B**ase (**DIB**) contains the addressing information needed by the local application. This specifies the network addresses and ACSE parameters of all the application processes on the network with which your application program will need to communicate. Depending on the type of operating system environment you use, the DIB takes various forms. Some OSI Stack vendors supply a network manager that contains a global DIB that all stacks (supplied by that vendor) can access to get the required addressing information. On other systems, as in the case of the SISCO OSI Stack, the DIB takes the form of a file that can be created using any text or word processor. On still other systems, there may be a special program such as a configuration utility that can be run to create the DIB file. Please consult your installation guide(s) for your specific system for more information.

4. MMS-EASE Configuration

The following steps are necessary to create and run an application:

1. Decide which MMS services are needed by your application program, and modify the **mmsop_en.h** file accordingly to create a service subset. See page 1-97 for more information on modifying this file.
2. Write and compile your program.
3. Compile **mmsop_en.c**.
4. Link your program with the **mmsop_en** object and the MMS-EASE libraries.
   MMS-EASE Core Library (**mms**)
   LLP Library (**mmsacse**)
   OSI Upper Layers Library (**osuil**)
   The C Runtime Library and Operating System support libraries.
5. Modify the **D**irectory **I**nformation **B**ase (**DIB**) per your requirements.
6. Start up the OSI Stack Please refer to your MMS-EASE product Installation Guide for more information.
7. Run your program.

MMS-EASE Demo Program

If you are using MMS-EASE for the first time, we strongly recommend that you take the time to build and execute the MMS-EASE demo program. See **Appendix F — MMS-EASE Demo Programs** for more information.

MMS Configuration Variables

The following variables are used to configure aspects of the MMS-EASE environment specific to MMS. These variables control configuration issues such as the maximum number of MMS channels, maximum number of outstanding requests, maximum files open, and maximum number of named variables. The larger these numbers are made, the more memory MMS-EASE allocates. All of these variables need to be set before calling **strt_MMS**.

```
extern ST_INT max_mms_chan = DEFAULT_NUM_MMS_CHAN;
```

This variable contains the maximum number of channels to be allocated for MMS-EASE. The default value is set to the constant, **DEFAULT_NUM_MMS_CHAN**. It can be decreased or increased depending on the particular OSI Stack implementation. The default is set in the include file, **mms_llp.h**.

```
extern ST_INT max_loc_open = MAX_FILES_OPEN;
```

This variable is used to determine how many open files the virtual machine will allow to be opened on the local system by remote nodes. This number is independent of limitations imposed by the operating system. See **Volume 3 — Module 9** for more information on files.

```
extern ST_INT m_loc_files_open;
```

This variable indicates the actual number of local files open.

```
extern ST_INT max_rem_open = MAX_FILES_OPEN;
```

This variable is used to determine how many open files the virtual machine will allow to be opened on remote systems. See **Volume 3 — Module 9** for more information on files.

```
extern ST_INT m_rem_files_open;
```

This variable indicates the actual number of remote files open.

extern ST_INT max_mmsease_types= 500;

This variable specifies the maximum number of named MMS types that the virtual machine will allow to be defined for the entire MMS-EASE environment. See **Volume 2 — Module 5** for more information on types.

extern ST_INT mms_type_count;

This variable keeps a running count of all the named types currently defined for the entire MMS-EASE environment.

extern ST_INT max_mmsease_vars = 500;

This variable specifies the maximum number of named MMS variables that the virtual machine will allow to be defined for the entire MMS-EASE environment. See **Volume 2 — Module 5** for more information on variables.

extern ST_INT mms_var_count;

This variable keeps a running count of all the named variables currently defined for the entire MMS-EASE environment.

extern ST_INT max_mmsease_nvlists = 500;

This variable indicates the maximum number of named variable lists allowed by the virtual machine to be defined locally. The default value for this number is 500.

extern ST_INT mms_nvlist_count;

This variable keeps a running count of all the named variable lists currently defined for the entire MMS-EASE environment.

extern ST_INT max_mmsease_doms;

This variable specifies the maximum number of named domains that the virtual machine will allow to be defined for the entire MMS-EASE environment. See **Volume 2 — Module 6** for more information on domains.

extern ST_INT max_mmsease_pis = 16;

This variable specifies the maximum number of program invocations that the virtual machine should allow to be defined for the entire MMS-EASE environment. See **Volume 2 — Module 6** for more information on program invocations.

extern ST_INT m_max_mods = 1;

This variable contains the maximum number of modifiers present in the **modifier_list** structure that the responder is willing to decode. See **Volume 3 — Module 7 — Modifier Handling** for more information on modifier handling. If more than this number of modifiers present is used, it will not be parsed, and a reject will be generated.

extern ST_INT mms_max_msgsize = DEFAULT_MAX_MSGSIZE;

This variable contains the maximum PDU message size to use. Even though MMS-EASE does not put any restrictions on the size of a PDU, the underlying operating system and/or the OSI Stack may limit the actual value that you can use.

NOTE: See also the description of the preferred initiate parameters on page 1-149 for information about the power up defaults for the preferred initiate parameters.

5. General Purpose Structures and Functions

This section deals with general purpose operation-specific data structures and general interface functions.

MMS Object Name Structure

OBJECT_NAME

This structure is used to specify any named MMS object. Examples of this may be a type name, variable name, or semaphore name. It is used in any MMS service that references a named MMS object. MMS objects are identified by name and by scope. The scope of an object may be either VMD-specific, domain-specific, or AA-specific (Application-Association-specific). For a domain-specific object, the name of the domain must be specified as well as the name of the object.

The following structure contains two versions. The compact form of the structure is used by MMS-EASE *Lite* only.

```
#if !defined (USE_COMPACT_MMS_STRUCTS)
struct object_name
  {
  ST_INT     object_tag;
  union
    {
    ST_CHAR vmd_spec [MAX_IDENT_LEN+1];
    ST_CHAR item_id  [MAX_IDENT_LEN+1];
    ST_CHAR aa_spec  [MAX_IDENT_LEN+1];
    } obj_name;
  ST_CHAR    domain_id [MAX_IDENT_LEN+1];
  SD_END_STRUCT
  };

#else        /* Use compact form */
struct object_name
  {
  ST_INT     object_tag;
  union
    {
    ST_CHAR *vmd_spec;
    ST_CHAR *item_id;
    ST_CHAR *aa_spec;
    } object_name;
  ST_CHAR    *domain_id;
  SD_END_STRUCT
  };
#endif
typedef struct object_name OBJECT_NAME;
```

Fields:

`object_tag` **VMD_SPEC**. This indicates the scope of the object is VMD-specific. Use the **vmd_spec** member of the union **obj_name**.

DOM_SPEC. This indicates the scope of the object is domain-specific. Use the **item_id** member of the union **obj_name** and the **domain_id** member of **OBJECT_NAME**.

AA_SPEC. This indicates the scope of the object is AA-specific. This means that it was previously defined over the network using a service request. Use the **aa_spec** member of the union **obj_name**.

`vmd_spec` This indicates the name of the VMD-specific object. It is used if **`object_tag = VMD_SPEC`**.

`item_id` This indicates the name of the domain-specific object. It is used if **`object_tag = DOM_SPEC`**. Use the **`domain_id`** member of **`OBJECT_NAME`** also.

`aa_spec` This indicates the name of the AA-specific object. It is used if **`object_tag = AA_SPEC`**.

`domain_id` This indicates the name of the specified domain. It is used if **`object_tag = DOM_SPEC`**. Use the **`item_id`** member of the **`OBJECT_NAME`** also.

Authentication STRUCTURE

`MDS_USERID`

This structure is used to specify a Directory Distinguished Name and an optional password. This structure is only used if the MMS-EASE application needs to be authenticated when binding to a X.500 DSA or a LDAP server.

```
typedef struct
  {
  ST_CHAR     name[MAX_DN_LEN + 1];
  ST_BOOLEAN passwd_pres;
  ST_CHAR     passwd[MAX_PSWD_LEN + 1];
  } MDS_USERID;
```

Fields:

`name` This character string contains the local Directory Distinguished Name to be used for authentication. This cannot be greater than the default of 255 specified in **`MAX_STAT_DN_LEN`** as defined in the **mds.h** include file.

`passwd_pres` **`SD_FALSE`**. passwd is not present.

`SD_TRUE`. passwd is present.

`passwd` This character string contains the password to be used for authentication. This cannot be greater than the default of 127 specified in **`MAX_STAT_PSWD_LEN`** as defined in the **mds.h** include file.

General Interface Functions

These functions are called by the user application program and are used to ”drive“ the MMS-EASE system under some operating systems. Certain operating systems require that functions of this class must be periodically called by the user to make sure that the OSI network board is being serviced. The contents of these functions are determined by MMS-EASE, and are not user-definable.

Start and End Functions

strt_MMS

Usage: This function is responsible for initializing the MMS-EASE system. It must be called before using any of the other MMS-EASE functions.

Function Prototype: `ST_RET strt_MMS (ST_VOID);`

Parameters: NONE

Return Value: `ST_RET`

`SD_SUCCESS`. No Error

`<> 0` Error Code. An error has occurred and the state change has failed. Supported errors may be displayed using **`ms_perror`**. See **Volume 3 — Module 11 — Error Handling** for an explanation of this function.

end_MMS

Usage: This function should be called before exiting to your operating system so that MMS-EASE may properly shut itself down.

Function Prototype: `ST_RET end_MMS (ST_VOID);`

Parameters: NONE

Return Value: `ST_RET`

`SD_SUCCESS`. No Error

`<> 0` Error Code. This means that an error has occurred and the state change has failed. Supported errors may be displayed using **`ms_perror`**. See **Volume 3 — Module 11 — Error Handling** for an explanation of this function.

Service Functions

The following interface functions provide the means of passing received indications and confirmations to the application program.

ms_comm_serve

Usage: This function must be called by the user's program to process messages received from the OSI Stack. This function, in turn, calls other MMS-EASE interface functions for such tasks as decoding messages, and pairing responses to requests. See NOTE below.

Function Prototype: `ST_RET ms_comm_serve (ST_VOID);`

Parameters: NONE

Return Value: `ST_RET`

`= 0`	Not Processed
`<> 0`	Processed

NOTES:

1. To process all outstanding events, you should call **`ms_comm_serve`** in a loop until **`SD_FALSE`** is returned.
2. **`ms_comm_serve`** services LLP events, indications and confirmations.

ms_service_check

Usage: This function is used by the client application to determine if a particular MMS service is supported according to it's subset and the set of services a server claims to support. The function examines the **mmsop_en** service array and examines the element in the array corresponding to the service for the bitmask **REQ_EN**. Support for the response is determined by examining the **service_resp** member of the **MMS_CHAN_INFO** structure.

Function Prototype:

```
ST_VOID ms_service_check (ST_INT chan,              ST_INT
                          service,                  ST_BOO-
                          LEAN *req_supported, ST_BOOLEAN
                          *resp_supported);
```

Input Parameters:

`chan` This contains the channel number of a **M_ACTIVE** channel.

`service` This is the opcode of the MMS Service checked for service support. Please refer to the **mmsop_en.h** file or **Volume I — Appendix B** for a list of valid opcodes.

Output Parameters:

`req_supported` This tells the application if the client request for the service was included in the MMS-EASE subset. **SD_TRUE** indicates support for the service.

`resp_supported` This tells the application if the server claims support for the service was included in the MMS-EASE subset. **SD_TRUE** indicates support for the service.

Return Value: `ST_VOID` (ignored)

Directory Authentication Functions

MMS-EASE supports simple Directory authentication. This means that you can instruct MMS-EASE to provide your name and password to the DSA when it binds to the Directory. This is accomplished by calling one of the following functions before calling **`strt_MMS`**.

If these functions are called before **`strt_MMS`**, MMS-EASE will relay authentication information to the Directory server. You do not have to call both functions.

mds_set_def_userid

Usage: This function is used to instruct MMS-EASE to set a user id and an optional password when using X.500 Directory Services while binding to the Directory.

Function Prototype:

```
ST_RET mds_set_def_userid (MDS_USERID *userid);
```

Parameters:

`userid` This is a pointer to the authentication control structure of type **`MDS_USERID`**. See page 1-86 for more information on this structure.

Return Value: `ST_RET`

`SD_SUCCESS`. Authentication was successful and Directory is bound.

`<> 0` Error Code is returned. Binding is rejected and the association will be terminated.

ldap_set_def_userid

Usage: This function is used to instruct MMS-EASE to set a user id and an optional password when using LDAP Directory Services while binding to the LDAP server.

Function Prototype: `ST_RET ldap_set_def_userid (MDS_USERID *userid)`

Parameters:

`userid` This is a pointer to the authentication control structure of type **`MDS_USERID`**. See page 1-86 for more information on this structure.

Return Value: `ST_RET`

`SD_SUCCESS`. Authentication was successful and Directory is bound.

`<> 0` Error Code is returned. Authentication failed, however Read operations on the Directory may still be performed.

Logging Function

u_mms_pdu_rcvd

Usage: This function pointer points to a function that MMS-EASE will invoke to examine all MMS PDUs generated or received by MMS-EASE. It also provides the capability of logging these PDUs.

For incoming PDUs, calls to this function are made just before the function to decode the PDU is invoked. Generally, this function is invoked followed by either a user indication, confirm, or error function from within a call to the **ms_comm_serve** function.

For outgoing PDUs, calls to this function are made as the PDU is sent to the LLP using a data transfer, associate request, or associate response. This occurs during a call to **mp_xxxx_resp** or **mv_xxxx_resp** functions.

Function Prototype:

```
extern ST_VOID (*u_mms_pdu_rcvd)(ST_INT chan,
                                 ST_INT type,
                                 ST_UINT ctxt,
                                 ST_INT pdu_len, ST_UCHAR
                                 *pdu);
```

Parameters:

chan	This contains the channel number over which MMS PDUs are sent and received.
type	This contains the LLP event code. It indicates the type of MMS PDU being sent or received. See the notes shown below.
ctxt	This mask selects the presentation context to be used from the list of negotiated Presentation Contexts.
pdu_len	This is the length (in bytes) of the data pointed by **pdu**.
pdu	This is the pointer to the PDU (in ASN.1-encoded form) to be examined or logged.

Return Value: ST_VOID (ignored)

NOTES: Below is a list of possible LLP Event Codes used in PDU logging to identify the types of PDUs either being received (INCOMING) or sent (OUTGOING):

INCOMING PDUs

ACSE_ASS_IND	0x0001	ACSE associate indications
ACSE_ASS_CFRM	0x0002	ACSE associate confirm
ACSE_RCV_DATA	0x0003	ACSE received data

OUTGOING PDUs

ACSE_SEND_DATA	0x0021	ACSE send data
ACSE_SEND_ASS_REQ	0x0022	ACSE send associate requests
ACSE_SEND_ASS_RESP	0x0023	ACSE send associate responses

Request and Indication Control Functions

These functions are called by the user's application program to perform various support functions required by MMS-EASE, specifically request and indication control.

ms_clr_ind_que

Usage: This function can be used during the processing of conclude indications or when terminating associations. It searches the internal communications queues for indications not yet responded to. It clears them by making them inactive. By using the **`all`** flag, this function can be instructed to only clear indications that have not yet been passed to the user's program using a call to the appropriate **`u_xxxx_ind`** function. It also can clear any indications that have been received for which a response has not yet been sent.

Function Prototype:

```
ST_RET ms_clr_ind_que (ST_INT chan, ST_BOOLEAN all);
```

Parameters:

`chan` This parameter indicates the channel number to clear. If the value is set to -1, this function clears the appropriate indications on all channels.

`all` **`SD_FALSE`**. This means clear only those indications that have not yet been passed to the user using a call to the corresponding **`u_xxxx_ind`** function.

`SD_TRUE`. This means clear **ALL** indications to which have not yet been responded.

Return Value: `ST_RET`

`SD_SUCCESS`. No Error.

`SD_FAILURE`. Error.

Data Structures Used: **`MMSREQ_IND`**

NOTE: As implied above, it is possible for an indication to remain unresponded to even though the **`u_xxxx_ind`** function has been called. In particular, the ObtainFile service requires that considerable processing be done before sending a response to the indication.

If it is possible that there are outstanding ObtainFile or other unresponded to indications, whether serviced by the virtual machine or not, **`ms_clr_ind_que`** should be called with **`all = SD_TRUE`**, when terminating an association.

ms_clr_mvreq

Usage: This function should be called at the end of user-defined virtual confirmation functions (**u_mv_xxxx_conf**). This allows MMS-EASE to free any data structures that the virtual machine created.

Function Prototype: `ST_VOID ms_clr_mvreq (MMSREQ_PEND *req);`

Parameters:

`req` This is a pointer to the request control data structure of type **MMSREQ_PEND** used for the request/confirmation and passed to the **u_mv_xxxx_conf** function.

Return Value: `ST_VOID`

Data Structures Used: **MMSREQ_PEND**

ms_count_ind_pend

Usage: This function is used to determine how many indications are currently outstanding for a specified channel.

Function Prototype: `ST_INT ms_count_ind_pend (ST_INT chan);`

Parameters:

`chan` This is the channel number for which the number of outstanding indications are to be determined. This must be a valid channel number between **0** and **`max_mms_chan -1`**.

Return Value:

`ST_INT`	`= 0`	no pending indications
	`<> 0`	number of indications pending for the selected channel

ms_count_req_pend

Usage: This function is used to determine how many requests are currently outstanding for a specified channel.

Function Prototype: `ST_INT ms_count_ind_pend (ST_INT chan);`

Parameters:

`chan` This is the channel number for which the number of outstanding requests are to be determined. This must be a valid channel number between **0** and **`max_mms_chan -1`**.

Return Value: `ST_INT`

`= 0`	no pending requests
`<> 0`	number of requests pending for the selected channel

6. Subset Creation

Since MMS-EASE is supplied in library form, it is easy to create applications that only use a subset of the supplied services. This allows programming without the code overhead of the unused functions. MMS-EASE library modules are divided by requester/responder classes and functionality.

To ensure that the application code size is kept to a minimum, please use the following steps. These steps will eliminate unused functions and create a MMS-EASE subset.

1. Make sure that your application code references only the functions required for your application.
2. Edit the file **mmsop_en.h**. A segment of this file is shown below. Enable only the MMS functionality required, by changing the definitions to enable or disable support for a particular MMS service. Responses and requests can be enabled or disabled independently. For example, if you want to disable a particular service such as the Status service, change the definition of the predefined constant, **MMS_STATUS_EN** to be equal to:
    a. **REQ_RESP_DIS** if you are not going to support this service, or
    b. **REQ_EN** if you are only going to support this service as a client, or
    c. **RESP_EN** if you are only going to support this service as a server, or
    d. **REQ_RESP_EN** if you are going to support this service both as a client and a server.

```
/*****************************************************************/
/* define the operation enable switches                          */
/*****************************************************************/
#define REQ_RESP_DIS         0x00   /* no support for req or resp  */
#define REQ_EN               0x01   /* support for request         */
#define RESP_EN              0x02   /* support for response        */
#define REQ_RESP_EN          0x03   /* support for resp and req    */

/*****************************************************************/
/* define the opcode enable switches                             */
/*****************************************************************/
#define MMS_INIT_EN          REQ_RESP_EN
#define MMS_CONCLUDE_EN      REQ_RESP_EN
#define MMS_CANCEL_EN        REQ_RESP_EN

#define MMS_STATUS_EN        REQ_RESP_DIS
#define MMS_USTATUS_EN       REQ_RESP_DIS
#define MMS_GETNAMES_EN      REQ_RESP_DIS
#define MMS_IDENT_EN         REQ_RESP_EN
```

3. Compile the **mmsop_en.c** file. This compilation changes the default values of some of the preferred initiate parameters, and some internal MMS-EASE variables. See page 1-149 for more information on preferred initiate parameters. This also eliminates internal references to the unused functions from the library.
4. **mmsop_en.c** file MUST be compiled with the **MAP30_ACSE** symbol defined.

    NOTE: Failure to compile with the **MAP30_ACSE** symbol defined will result in an error reported by the linker.

5. When linking your programs with the MMS-EASE libraries, the **mmsop_en** object must be linked with the libraries and your application's object code.

    This process prevents all unnecessary MMS-EASE code from being included in your application.

7. Logging Facilities

MMS-EASE contains a logging system, referred to as the **S_LOG** (**S**ISCO **Log**ging) system. This system provides a flexible and useful approach to system logging, and is easily expanded to meet the logging requirements of most end user applications.

General Logging

Below is a list of features available in the general **S_LOG** system:

- Logging data is accepted in **printf** type format.
- Hex buffers are logged.
- Continuation (multi-line messages) is supported.
- Information is time stamped. The options are either by Date and Time (e.g., Tue Jun 13 15:57:32 1995) or elapsed (millisecond resolution) timing can be used.
- **S_LOG** allows the capability of using multiple logging control elements with one log file per logging control element.
- It provides the capability to include Source file and Line Number information for debugging. '
- In-memory logging is available for profiling timing information

File Logging

Below is a list of features available in the **S_LOG** file system.

- **S_LOG** logs to circular file.
- It allows dynamic enabling and disabling of file logging using the supplied functions.
- Other options :

Memory Logging

Below is a list of features available in the **S_LOG** memory system.

- **S_LOG** logs to a list of memory resident buffers for collection of log information in real time. Buffers are accessible to the application and can be written to file under program control.

Log Control Data Structure

This structure is used to set logging control flags including file and memory logging control. In addition it contains bit masked variables that can be used by the application to determine whether an item is to be logged.

```
typedef struct log_ctrl
  {
  ST_UINT32  logCtrl;
  FILE_LOG_CTRL     fc;
  MEM_LOG_CTRL      mc;
/* Application specific information                              */
  ST_UINT32  logMask1;
  ST_UINT32  logMask2;
  ST_UINT32  logMask3;
  ST_UINT32  logMask4;
  ST_UINT32  logMask5;
  ST_UINT32  logMask6;
  } LOG_CTRL;
```

Fields:

`logCtrl` A mask of bits that determine the type or types of logging desired. These bits can be ORed together to form any combination. Acceptable values are:

`LOG_MEM_EN (0x0001L)`	Enables Memory Logging
`LOG_FILE_EN  (0x0002L)`	Enables File Logging
`LOG_TIME_EN  (0x0008L)`	Time stamping is enabled. One of the following bits must also be set.
`LOG_TIMEDATE_EN (0x0010L)`	Use **`ctime`** to create the time stamp string.
`LOG_DIFFTIME_EN (0x0020L)`	Use system specific services to log the number of milliseconds that have elapsed since logging was started.

`fc` This structure of type **`FILE_LOG_CTRL`** contains the control information for file logging. This is used if the logCtrl bit **`LOG_FILE_EN`** is set. See the next sections for more information.

`mc` This structure of type **`MEM_LOG_CTRL`** contains the control information for memory logging. This is used if the logCtrl bit **`LOG_MEM_EN`** is set. See the next sections for more information.

`logMask1...6` These are available for use by the application to determine whether an item is to be logged. Using these masks, you will have 192 bits available for setting various log levels. The application would normally reference these logmasks in a C MACRO. The following example shows the simplest approach for **S_LOG** integration into an existing system.

Using the S_LOG Logmasks

This section describes how to use the **`logMask1...6`** capabilities of **S_LOG**.

```
/************************************************************************/
/* For debug version, use a static pointer to avoid duplication of      */
/* __FILE__ strings.                                                    */
static ST_CHAR *thisFileName = __FILE__;
```

In the following example **`SLOG1_1`** is used as a way to send application specific error messages with one data item to the log file. The application code might look like:

```
PR_Log_Err( "Hard error detected %d", errno );
```

and the macro **PR_Log_Err** might be defined as follows:

```
#define PR_LOG_ERR          0x00000001L
#define PR_ERR_LT           0
#define PR_Log_Err1(a,b) SLOG1_1(sLogCtrl,PR_LOG_ERR,PR_ERR_LT,a,b)
```

The macro for **SLOG1_1** is found in the header file **slog.h** and is defined as follows::

```
#define SLOG1_1(lc,mask,id,a,b) {\
  if (lc->logMask1 & mask)\
  slog (sLogCtrl,id, thisFileName,__LINE__,a,b);\
  }
```

S_LOG macros found in **slog.h** follow the naming convention: SLOG**x**_**y**, where **x** indicates which of the 6 logmasks to AND with the log mask, **y** denotes the number of data elements to use with the format specifier (**a**). For example, because the SLOG macro listed below examines log mask 2 and passes three data items to be written to the log format specifier, it is called **SLOG2_3**.

```
define SLOG2_3(lc,mask,id,a,b,c,d) {\
    if (lc->logMask2 & mask)\
    slog (sLogCtrl,id, thisFileName,__LINE__,a,b,c,d);\
    }
```

Using log masks is not the only way for the application to call **S_LOG**. The application may use a different MACRO convention. As a comparison, MMS-EASE uses global variables to determine when it should call **S_LOG** functions. It does not use **logMask1...6** as is shown in the example below.

```
#define MLOG_DEC2(a,b,c) {\
    if (mms_debug_sel & MMS_LOG_DEC)\
    slog (sLogCtrl,MMS_LOG_DEC_TYPE,\
    thisFileName,__LINE__,a,b,c);\
    }
```

File Control Data Structure

This structure is used to set logging control information for file logging.

```
typedef struct file_log_ctrl
  {
  ST_ULONG  maxSize;
  ST_CHAR   *fileName;
  ST_UINT   ctrl;
  ST_UINT   state;  /* DO NOT USE       */
  FILE       *fp;         /* DO NOT USE       */
  } FILE_LOG_CTRL;
```

Fields:

`maxSize` This indicates the maximum size of the log file when file wrap is enabled (default is 1MB).

`fileName` This is a pointer to the log file name. Default name is **mms.log**.

`ctrl` These are file logging control flags. The following are control bits used to enable and disable the file logging options. These bits can be ORed together to form any combination. Acceptable values are:

`FIL_CTRL_WIPE_EN (0x0001)`	Enables the use a **wipe bar** to show where the current data is in a wrapped file.
`FIL_CTRL_WRAP_EN (0x0002)`	Enables wrapping of the file. Note that file wrapping is temporarily disabled during a hex dump.
`FIL_CTRL_MSG_HDR_EN (0x0004)`	Enables a message header to be displayed when the file is written.

`FIL_CTRL_NO_APPEND (0x0008)`	When first opening the log file, the existing contents are destroyed.
`FIL_CTRL_HARD_FLUSH (0x0010)`	Close and reopen the log file after each write. This should be used to better ensure not losing any log data if there is a crash.
`FIL_CTRL_SETBUF_EN (0x0020)`	Enables the use of the **`setbuf(fh,NULL)`** command to turn off buffering. For some compilers, this will slow application processing down but should be used to better ensure not losing log data if there is a crash.

```
state       /*      For Internal SISCO Use — Do Not Use     */
fp          /*      For Internal SISCO Use — Do Not Use     */
```

Memory Control Data Structure

This structure is used to set logging control flags for memory logging.

```
typedef struct mem_log_ctrl
  {
  ST_INT       maxItems;
  ST_CHAR      *dumpFileName;
  ST_UINT      ctrl;
  ST_UINT      state;           /* DO NOT USE       */
  LOGMEM_ITEM *item;            /* DO NOT USE       */
  ST_INT       nextPut;         /* DO NOT USE       */
  } MEM_LOG_CTRL;
```

Fields:

`maxItems` This indicates the maximum numbers of items to allocate at powerup.

`dumpFileName` This is a pointer to the file name of the memory dump.

`ctrl` These are memory logging control flags. The following are control bits used to enable and disable the memory logging options. These bits can be ORed together to form any combination. Acceptable values are:

`MEM_CTRL_MSG_HDR_EN (0x0001)`	Enables a message header to be displayed when the file is written.
`MEM_CTRL_AUTODUMP_EN (0x0002)`	Enables autodump of memory buffers.
`MEM_CTRL_HEX_LOG (0x0004)`	Enables memory logging in hexadecimal.

```
state       /*      For Internal SISCO Use — Do Not Use     */
item        /*      For Internal SISCO Use — Do Not Use     */
nextPut     /*      For Internal SISCO Use — Do Not Use     */
```

S_LOG Global Variables and Constants

The following variables are used with SISCO Logging:

`extern ST_INT sl_max_msg_size = MAX_LOG_SIZE;`

This variable contains the maximum message size of a **S_LOG** message. The default value is set to the constant, **`MAX_LOG_SIZE`**. The default is set in the include file, **slog.h** to a value of 500 bytes.

`extern ST_CHAR slogTimeText[TIME_BUF_LEN];`

This variable is used to create time strings for **S_LOG**ging. The maximum size of the buffer **`TIME_BUF_LEN`** is defined as a default to be 30.

`#define SLOG_MEM_BUF_SIZE`

This constant represents the maximum line length of a memory resident message. Messages longer than this constant supplied to the **`slogMem`** function are truncated at this limit. The default is 125 characters.

Initializing S_LOG

To use **S_LOG**, a **`LOG_CTRL`** data structure must be created and initialized, and the MMS-EASE global variable **`sLogCtrl`** set to point to the structure. Be sure to zero out all internal fields in the structure: **`fc.state`** and **`fc.fp`** for file logging and **`mc.state`**, **`mc.item`**, and **`mc.nextPut`** for memory logging. In case the application does not create a **`LOG_CTRL`** structure, the default structure will be used.

It is defined as follows:

```
logCtrl = LOG_FILE_EN | LOG_TIMEDATE_EN;
fc.maxSize = 1000000;
fc.fileName = "mms.log";
fc.ctrl = FIL_CTRL_WIPE_EN | FIL_CTRL_WRAP_EN | FIL_CTRL_MSG_HDR_EN;
```

NOTE: A sample user function for initializing the **S_LOG** system is provided in the sample source module **mmsamisc.c**. This function, **`m_set_log_cfg`**, reads the desired logging configuration from a text file called **mms_log.cfg**, and sets up logging. This file is supplied with the MMS-EASE demo program. Please refer to **Appendix F — MMS-EASE Demo Programs** for more information on this file.

S_LOG Functions

The following functions are used perform application level logging.

slog

Usage: This function is the general purpose logging function. It takes care of both memory and file logging as required.

Function Prototype:

```
ST_VOID slog (LOG_CTRL *lc,
              ST_INT logType,
              ST_CHAR *sourceFile,
              ST_INT lineNum,
              ST_CHAR *format, ...);
```

Parameters:

`lc`	This is a pointer to the log control structure of type **`LOG_CTRL`**. See page 1-100 for more information on this structure.
`logType`	This is the log type identifier used to indicate the log message class. The purpose of the **`logType`** is to place some arbitrary number next to the message in the log file. When dealing with large log files, choosing the number carefully makes it easy to find the message using the search feature of a text editor.
`sourceFile`	This is a pointer to the name of the source file containing the call to slog. It is used when logging debug information indicating which C file received the log message. It may be passed a NULL argument if this information is unwanted.
`lineNum`	This indicates the source file line number if a source file argument is passed in as a non-NULL value. The typical way to determine the line number of a C program is to use the built-in preprocessor command __**`LINE`**__.
`format`	This is a pointer to the optional **`printf`** type message to log.

Return Value: `ST_VOID` (ignored)

slogHex

Usage: This function is the Hexadecimal data logging function. It takes care of both memory and file logging as required.

Function Prototype:

```
ST_VOID slogHex (LOG_CTRL *lc,                    ST_INT
                 logType,                         ST_CHAR
                 *fileName,                       ST_INT
                 lineNum,                         ST_INT
                 numBytes,                        ST_VOID
                 *hexData);
```

Parameters:

`lc` This is a pointer to the log control structure of type **`LOG_CTRL`**. See page 1-100 for more information on this structure.

`logType` This is the log type identifier used to indicate the log message class. The purpose of the **`logType`** is to place some arbitrary number next to the message in the log file. When dealing with large log files, choosing the number carefully makes it easy to find the message using the search feature of a text editor.

`sourceFile` This is a pointer to the name of the source file containing the call to slog. It is used when logging debug information indicating which C file received the log message. It may be passed a null argument if this information is unwanted.

`lineNum` This indicates the source file line number if a source fileargument is passed in as a non-null value. The typical way to determine the line number of a C program is to use the built-in preprocessor command **`__LINE__`**.

`numBytes` This indicates the number of bytes to log.

`hexData` This is a pointer to a data buffer that is logged in hexadecimal format.

Return Value: `ST_VOID` (ignored)

slogCloneFile

Usage: This function is used to copy the contents of a log file to a new file name. The source log file is supplied in the **LOG_CTRL** information. The new file name is supplied in the second argument. When the source log file is open and being used by the **S_LOG** susbsystem, it is closed, copied, and reopened to its prior location before the function returns.

Function Prototype:

```
ST_VOID slogCloneFile (LOG_CTRL *lc, ST_CHAR *newfile);
```

Parameters:

`lc` This is a pointer to the log control structure of type **LOG_CTRL**. The **lc->fc.fileName** is the name of the source file name. See page 1-100 for more information on this structure.

`newfile` This is a pointer to a string containing the new file name.

Return Value: `ST_VOID` (ignored)

slogCloseFile

Usage: This function closes the file being used for logging. The next item logged will cause the file log to be re-initialized.

Function Prototype: `ST_VOID slogCloseFile (LOG_CTRL *lc);`

Parameters:

`lc` This is a pointer to the log control structure of type **`LOG_CTRL`**. See page 1-100 for more information on this structure.

Return Value: `ST_VOID` (ignored)

slogGetMemCount

Usage: This function returns the number of used memory resident message buffers when memory slogging is in use.

Function Prototype: `ST_INT slogGetMemCount (LOG_CTRL *lc);`

Parameters:

`lc` This is a pointer to the log control structure of type **`LOG_CTRL`**. See page 1-100 for more information on this structure.

Return Value: `ST_INT` This returns the number of memory buffers containing slog messages.

slog_dyn_log_fun

Usage: This function pointer can be set to point to a function in the application which is called each time information is sent to **slog** or **slogHex**. This mechanism allows the application to process log data in a manner not available in the **S_LOG** system

Function Prototype:

```
extern ST_VOID (*slog_dyn_log_fun) (LOG_CTRL *lc,   ST_INT
                                    logType, ST_CHAR
                                    *sourceFile, ST_INT
                                    lineNum, ST_INT bu-
                                    fLen, ST_CHAR *buf);
```

Parameters:

lc	This is a pointer to the log control structure of type **LOG_CTRL**. See page 1-100 for more information on this structure.
logType	This is the log type identifier used to indicate the log message class. The purpose of the **logType** is to place some arbitrary number next to the message. When dealing with large quantities of information choosing the number carefully makes it easy to see the message.
sourceFile	This is a pointer to the name of the source file containing the call to slog. It allows the application to know which C file called **slog** or **slogHex**. It may be received as a null argument if this information is intentionally not given or unknown.
lineNum	This indicates the source file line number if a source file argument is passed in as a non-null value. The typical way to determine the line number of a C program is to use the built-in preprocessor command **__LINE__**.
bufLen	This is the length of the string being sent to the log file.
buf	This is a pointer to the information buffer.

Return Value: ST_VOID (ignored)

NOTE: The sample source module , **mmsamisc.c**, has an example of how to use 'dynamic' logging. Refer to the functions **do_debugset** and **screenLogFun** for an example that displays the log information to the screen or to a file using the **mms_debug_log** stream.

slog_service_fun

Usage: This function pointer may be set to point to a function in the application that is called periodically during slow **S_LOG** operations such as cloning a file. The intention of this function is to allow a real-time application processing time while **S_LOG** has been transferred control of the processor. File logging is temporarily disabled when this function is called.

Function Prototype:
```
extern ST_VOID (*slog_service_fun) (ST_VOID);
```

Parameters: None

Return Value: `ST_VOID` (ignored)

MMS-EASE Log Levels

The amount of logging produced by MMS-EASE is controlled by setting global MMS-EASE variables.

The global variables used to control the log levels are: **mms_debug_sel**, **asn1_debug_sel**, **s_debug_sel**, **chk_debug_en**, and **list_debug_sel**.

These variables hold log control bits for enabling and disabling the various levels of logging as shown below.

```
extern ST_ULONG mms_debug_sel;
```

CONSTANT	BIT ASSIGNMENTS	DESCRIPTION
MMS_LOG_DEC	0x00000001L	Enable logging of MMS decoding process.
MMS_LOG_ENC	0x00000002L	Enable logging of MMS encoding process.
MMS_LOG_ACSE	x00000004L	Enable logging of ACSE transaction messages.
MMS_LOG_LLC	0x00000008L	Enable logging of LLC transaction messages.
MMS_LOG_INDQUE	0x00000010L	Enable logging of Indication queue transactions.
MMS_LOG_REQQUE	0x00000020L	Enable logging of Request queue transactions.
MMS_LOG_VM	0x00000100L	Enable logging of Virtual Machine Operations.
MMS_LOG_DATA	x00000200L	Enable logging of all Data Conversion errors.
MMS_LOG_ERR	0x00010000L	Enable logging of abnormal errors.
MMS_LOG_NERR	x00020000L	Enable logging of normal errors.
MMS_LOG_PDU	0x00040000L	Enable logging of all MMS PDUs.
MMS_LOG_RT	0x00010000L	Enable logging of all RunTime Type transactions.
MMS_LOG_RTAA	x00020000L	Enable logging of all RunTime Type AlternateAccess transactions.
MMS_LOG_AA	0x00040000L	Enable logging of all Alternate Access transactions.
MMS_LOG_REQ	0x01000000L	Enable logging of all MMS Requests.
MMS_LOG_RESP	x02000000L	Enable logging of all MMS Responses.
MMS_LOG_IND	0x04000000L	Enable logging all Indications.
MMS_LOG_CONF	x08000000L	Enable logging all Confirmations.

The following defines are user-reserved. These are not used by MMS-EASE

MMS_LOG_USR_IND	0x00000100L	Enable logging of User Indications.
MMS_LOG_USR_CONF	0x00000200L	Enable logging of User Confirmations.

By default, **mms_debug_sel** is set to **MMS_LOG_ERR**.

```
extern ST_UINT asn1_debug_sel;
```

CONSTANT	BIT ASSIGNMENTS	DESCRIPTION
ASN1_LOG_DEC	x0001	Enable logging of ASN.1 decode process.
ASN1_LOG_ENC	x0002	Enable logging of ASN.1 encode process.
ASN1_LOG_ERR	x0004	Enable logging of abnormal ASN.1 errors.
ASN1_LOG_NERR	x0008	Enable logging of normal ASN.1 errors.

By default, **asn1_debug_sel** is set to **ASN1_LOG_ERR**.

```
extern ST_UINT s_debug_sel;
```

CONSTANT	BIT ASSIGNMENTS	DESCRIPTION
ACSE_IND_PRINT	0x0001	Enable logging of upbound SUIC transactions.
ACSE_CNF_PRINT	0x0002	Enable logging of downbound SUIC transactions.
ACSE_ERR_PRINT	0x0004	Enable logging of abnormal SUIC errors.
ACSE_DEC_PRINT	0x0008	Enable logging of SUIC decode process.
ACSE_NERR_PRINT	0x0040	Enable logging of normal SUIC errors.

~~By default, **s_debug_sel** is set to **ACSE_ERR_PRINT**.~~

```
extern ST_BOOLEAN list_debug_sel;
```

Setting this variable to **SD_TRUE** causes all internal MMS-EASE list operations to be logged. By default, **list_debug_sel** is not set.

```
extern ST_UINT chk_debug_en;
```

MEM_LOG_ERR	0x0001	Enable logging of abnormal memory errors.
MEM_LOG_MALLOC	0x0002	Enable logging of **chk_malloc** calls.
MEM_LOG_CALLOC	0x0004	Enable logging of **chk_calloc** calls.
MEM_LOG_REALLOC	0x0008	Enable logging of **chk_realloc** calls.
MEM_LOG_FREE	x0010	Enable logging of **chk_free** calls.

By default, **chk_debug_en** is set to **MEM_LOG_ERR**.

8. Linked List Manipulation

MMS-EASE provides a set of data structures and functions that allows access to a circular doubly linked list. You can use these functions in your application.

Link List Data Structure

In order to use the MMS-EASE list functions, you must create a data structure that contains the following data structure as its first element. This allows you to use one set of list manipulation primitives to be used with any structure containing it.

```
typedef struct dbl_lnk
  {
  struct dbl_lnk *next;
  struct dbl_lnk *prev;
  } DBL_LNK;
```

Fields:

`next` This points to the next element in the linked list.

`prev` This points to the previous element in the linked list.

Functions for Generic Link List Handling

list_get_first

Usage: This function is used to unlink the first element from a list and return its address.

Function Prototype: `ST_VOID *list_get_first (DBL_LNK **first_el);`

Parameters:

`first_el` This is a pointer of type **`DBL_LNK`** to the address of the head of a list pointer.

Return Value: A pointer to the unlinked element.

list_get_next

Usage: This function is used to traverse a circular doubly linked list from the beginning to the end using the next **DBL_LNK** structure member.

Function Prototype:

```
ST_VOID *list_get_next (DBL_LNK *list_head,          DBL_LNK
                                    *next_el);
```

Parameters:

`list_head`	This is a pointer of type **DBL_LNK** to the address of the head of a list.
`next_el`	This is a pointer of type **DBL_LNK** to the current element in the list.

Return Value: This is the pointer to the next node element in the list. When the next element in the list is the head of the list pointer, then the function returns a null value.

list_unlink

Usage: This function is used to unlink an element from a circular doubly linked list.

Function Prototype:
```
ST_RET list_unlink (DBL_LNK **list_head,
                    DBL_LNK *unlink_el);
```

Parameters:

`list_head`	This is a pointer of type **`DBL_LNK`** to the address of the head of a list.
`unlink_el`	This is a pointer of type **`DBL_LNK`** to the element to be unlinked from the list.

Return Value: `ST_RET`

`SD_FAILURE`.	The element is not present in the list, or bad parameter.
`SD_SUCCESS`.	The element was found in the list and unlinked.

list_add_first

Usage: This function is used to add an element as the first element to a circular doubly linked list.

Function Prototype:
```
ST_RET list_add_first (DBL_LNK **list_head, DBL_LNK *first_el);
```

Parameters:

`list_head` This is a pointer of type **`DBL_LNK`** to the address of the head of a list pointer.

`first_el` This is a pointer of type **`DBL_LNK`** to element to be added to the list.

Return Value: `ST_RET`

`SD_FAILURE`. The element was not added to the front of the list. The old state of the list is preserved.

`SD_SUCCESS`. The element was added to the beginning of the list. The pointer to the head of the list (**`list_head`**) has been modified.

list_add_last

Usage: This function is used to add an element as the last element to a circular doubly linked list.

Function Prototype:
```
ST_RET list_add_last (DBL_LNK **list_head, DBL_LNK *last_el);
```

Parameters:

`list_head` This is a pointer of type **DBL_LNK** to the address of the head of a list pointer.

`last_el` This is a pointer of type **DBL_LNK** to element to be added to the list.

Return Value: `ST_RET`

SD_FAILURE. The element was not added to the back of the list. The old state of the list is preserved.

SD_SUCCESS. The element was added to the end of the list. The pointer to the head of the list (**list_head**) has been modified if this was an empty list.

list_add_first

Usage: This function is used to add an element as the first element to a circular doubly linked list.

Function Prototype:
```
ST_RET list_add_first (DBL_LNK **list_head, DBL_LNK *first_el);
```

Parameters:

`list_head` This is a pointer of type **`DBL_LNK`** to the address of the head of a list pointer.

`first_el` This is a pointer of type **`DBL_LNK`** to element to be added to the list.

Return Value: `ST_RET`

`SD_FAILURE`. The element was not added to the front of the list. The old state of the list is preserved.

`SD_SUCCESS`. The element was added to the beginning of the list. The pointer to the head of the list (**`list_head`**) has been modified.

list_move_to_first

Usage: This function is used to unlink an element from where ever it is present in the list and add it as the first element of a second linked list.

Function Prototype:

```
ST_RET list_move_to_first (DBL_LNK **list_head,     DBL_LNK
                           **next_head,    DBL_LNK
                           *first_el);
```

Parameters:

`list_head` This is a pointer of type **`DBL_LNK`** to the address of the head of a list pointer.

`next_head` This is a pointer of type **`DBL_LNK`** to the address of the head of a next list pointer.

`first_el` This is a pointer of type **`DBL_LNK`** to element to be moved from the first list and added to the list.

Return Value: `ST_RET`

`SD_FAILURE`. The element was not moved from the first list to the second list. The unlink step has failed, so the old state of the first list is preserved.

`SD_SUCCESS`. The element was unlinked from the first list and added to beginning of the second list.

list_find_node

Usage: This function is used to verify that a node is linked in as a member of a linked list.

Function Prototype:
```
ST_RET list_find_node (DBL_LNK *list_head,
                       DBL_LNK *first_el);
```

Parameters:

`list_head`	This is a pointer of type **`DBL_LNK`** to the address of the head of a list pointer.
`first_el`	This is a pointer of type **`DBL_LNK`** to element to be verified.

Return Value: `ST_RET`

`SD_FAILURE`. The node was not found in the list.

`SD_SUCCESS`. The node was present in the list.

list_add_node_after

Usage: This function is used to add a node to the list.

Function Prototype:
```
ST_RET list_add_node_after (DBL_LNK *cur_node,
                            DBL_LNK *new_node);
```

Parameters:

`cur_node` This is a pointer of type **DBL_LNK** that represents the location in the list after which to add the **new_node**.

`new_node` This is a pointer of type **DBL_LNK** to the node that is added to the list.

Return Value: `ST_RET`

SD_FAILURE. The **new_node** was not added to the list.

SD_SUCCESS. The **new_node** was added to the list.

list_get_sizeof

Usage: This function is used to get the size of a circular doubly linked list.

Function Prototype: `ST_INT list_get_sizeof (DBL_LNK *list_head_pointer);`

Parameters:

`list_head_pointer`	This is a pointer of type **`DBL_LNK`** to the head of a list.

Return Value: `ST_INT`

`= 0`	The list is empty.
`<>0`	Returns the number of elements in the linked list.

9. Memory Management Tools

MMS-EASE provides a set of memory management tools that include logging and integrity checking. To do so, replacements for the standard C runtime library functions **malloc**, **calloc**, **realloc**, and **free** are provided. These replacements are **chk_malloc**, **chk_calloc**, **chk_realloc**, and **chk_free**, respectively. These functions accept the same arguments as their counterparts from the standard C runtime library and are used internally by MMS-EASE. The functions are exposed so that MMS-EASE application s can take advantage of their features. The MMS-EASE memory management tools have the following features:

1. Every time **chk_free** is called to free a pointer that was not returned by **chk_calloc**, **chk_malloc**, or **chk_realloc**, an error message is logged. This can be helpful to determine the following problems::

   a. The application was freeing an invalid pointer.

   b. The application was freeing the same pointer more than once.

   c. The application was freeing a null pointer.

2. A pointer passed to the **chk_free** function is set to an illegal value. This can be helpful to determine the following problem:

   a. The application was freeing a pointer but using the pointer at a later point in the program.

3. If the application uses a lot of memory and eventually is running out, the functions **chk_calloc**, **chk_malloc**, and **chk_realloc** will detect this condition and log all the pointers currently under the view of the tools and report this error using a function pointer. This can be helpful in finding the following problems:

   a. The application is running out of memory because it is allocating the memory but not giving it back.

   b. The application is overwriting a portion of dynamic memory and corrupting the C runtime library memory management list.

4. Calling the function **dyn_mem_ptr_status** will log a current list of allocated pointers. This can be helpful in finding the following problems:

   a. If the list continues to grow, the application is probably allocating memory but not giving it back.

   b. If **dyn_mem_ptr_status** crashes in the middle of displaying information, the memory list has been corrupted before that point. In this situation, it is helpful to insert temporary calls in the program to **chk_mem_list**. The calls to the memory list validation tool may help you zero in on the program logic which is causing the problem.

COMPILING AND LINKING THE TOOLS

If you decide to use the SISCO memory management tools in your application, it is recommended that you define the **DEBUG_SISCO** symbol when compiling your program. ~~This symbol can be used in conjunction with the **S_THISFILE** symbol shown as follows:~~

```
#ifdef DEBUG_SISCO
#ifdef S_THISFILE
#define chk_realloc(x,y)  x_chk_realloc (x,y,thisFileName,__LINE__)
#define chk_calloc(x,y)   x_chk_calloc  (x,y,thisFileName,__LINE__)
#define chk_malloc(x)     x_chk_malloc  (x,thisFileName,__LINE__)
#define chk_free(x)       x_chk_free_wipe ((ST_VOID
**)&(x),thisFileName,__LINE__)
#else
#define chk_realloc(x,y)  x_chk_realloc (x,y,(ST_CHAR *) __FILE__,__LINE__)
#define chk_calloc(x,y)   x_chk_calloc  (x,y,(ST_CHAR *) __FILE__,__LINE__)
#define chk_malloc(x)     x_chk_malloc  (x, (ST_CHAR *) __FILE__,__LINE__)
#define chk_free(x)       x_chk_free_wipe ((ST_VOID **)&(x),  (ST_CHAR *)
__FILE__,__LINE__)
#endif
#else   /* !DEBUG_SISCO */
.
.
.
#endif
```

If you define **S_THISFILE**, you must define a static variable, **thisFilename**, in all of your modules that include **mem_chk.h** as follows:

```
#ifdef DEBUG_SISCO
static char *thisFileName = __FILE__;
#endif

#include "mem_chk.h"
```

Memory Allocation Global Variables

The following variables are used with the SISCO Memory Allocation Tools:

extern ST_VOID *m_bad_ptr_val;

This value of this variable is written to all pointers passed to **chk_free**. Changing the value of the pointer is done to cause an immediate runtime crash should a released pointer be used later in the program. The default value is `(ST_VOID *) -1`.

extern ST_BOOLEAN m_check_list_enable;

This variable is used to enable list validation and overwrite checking on every alloc and free call. When the application experiences random crashes, enabling this feature is highly recommended. The default is **SD_FALSE**.

extern ST_BOOLEAN m_find_node_enable;

This variable is used to enable searching the memory list for the element before accessing the memory during **chk_realloc** and **chk_free** calls. The value of **SD_TRUE** enables searching the memory list. The value of **SD_FALSE** disables the search and may speed up the application. ~~The default is **SD_TRUE**.~~

extern ST_BOOLEAN m_no_realloc_smaller;

This variable will cause chk_realloc not to realloc when the new size is smaller than the old size. Not reallocating a buffer to a smaller size is desirable on systems whose memory management algorithms lead to excessive fragmentation. The default is **SD_FALSE**.

extern ST_CHAR *m_pad_string;

This is a pointer to string of octets which are placed as a header and footer around the actual contents of the buffer. When **m_check_list_enable** is set to **SD_TRUE** the value in this string must be present as the header and footer each time the buffer is validated or the memory error function pointer ***mem_chk_err** will be invoked. The default value of the string is `0xDEADBEEF`.

extern ST_INT m_num_pad_bytes;

This variable indicates the number of bytes in the **m_pad_string**. The default is 4 bytes.

extern ST_BOOLEAN m_fill_en;

This variable is used to enable a feature which will fill up a freed buffer with values that may cause the program to crash should references to locations within the buffer still be active after the buffer has been freed. When set to **SD_TRUE** the value of the **m_fill_byte** is written to each byte in a buffer freed by calling **chk_free**. The default is **SD_FALSE**.

extern ST_UCHAR m_fill_byte;

This variable contains the value that is written to buffers freed when **m_fill_en** is set to **SD_TRUE**. The default is `0xCC`.

extern ST_BOOLEAN m_mem_debug;

This variable must be set to **SD_TRUE** to enable any of the memory tool validation features. Setting this value to **SD_FALSE** causes all memory validation code to be circumvented and calls to **chk_calloc**, **chk_malloc**, **chk_realloc**, and **chk_free** essentially map on to the C runtime library with little or no overhead. The default is **SD_TRUE**.

Functions for dynamic memory allocation

dyn_mem_ptr_status

Usage: This function will log the current list of allocated pointers to a file attached to **S_LOG** subsystem. The information contains the size of each buffer, the file and line where the buffer was allocated, statistics on how many pointers are allocated, and how much total dynamic memory is in use.

Function Prototype: `ST_VOID dyn_mem_ptr_status (ST_VOID);`

Parameters: None

Return Value: `ST_VOID` (ignored)

dyn_mem_ptr_statistics

Usage: This function is used to display statistics associated with the dynamic memory heap. The four pieces of information shown are :

1. The total number of pointers allocated.
2. the total amount of memory allocated,
3. the maximum number of pointers allocated, and
4. the maximum amount of memory allocated.

Unless the program is not releasing the dynamic memory it allocates using the memory management tools, the maximum values will be greater than the total values. Maximum, in this case, refers to the values accumulated in the tools since the first buffer was allocated.

Function Prototype:

```
ST_VOID dyn_mem_ptr_statistics (ST_BOOLEAN log_to_screen);
```

Parameters:

`log_to_screen`

`SD_TRUE`. Dynamic memory statistics are shown to the screen.

`SD_FALSE`. Dynamic memory statistics will be logged to the file attached to the **S_LOG** subsystem.

Return Value: `ST_VOID` (ignored)

check_mem_list

Usage: This function will check the integrity of the memory heap associated with the **chk**_ family of functions. Pointers in the heap are validated and traversed to verify that the list is intact. The memory buffer headers and footers are checked to catch memory overwrite problems. Although this function can be called from anywhere in the application to catch an overwrite, setting the global variable **m_check_list_enable** to **SD_TRUE** will cause this function to be called by the **chk**_ functions each time they are used. Any error detected by this function is reported by calling the **mem_chk_err** function pointer. To be of any use, the **mem_chk_err** function pointer should be set to point to a function in the application that displays the error or logs it to a file.

Function Prototype:

```
ST_VOID check_mem_list (ST_VOID);
```

Parameters: None

Return Value: ST_VOID (ignored)

chk_alloc_ptr

Usage: This function will verify that the pointer passed to this function is on the memory management list and when **m_check_list_enable** is set to **SD_TRUE**, header footer checking is performed on the buffer. If the pointer or buffer was in error, the **mem_chk_err** function pointer will be invoked.

Function Prototype: `ST_RET chk_alloc_ptr (ST_VOID *ptr);`

Parameters:

`ptr`	This is a pointer to a dynamically allocated memory buffer.

Return Value: `ST_RET`

SD_SUCCESS. This means the buffer was OK.

SD_FAILURE. This means the pointer or buffer was corrupted.

chk_malloc

Usage: This function replaces the standard C malloc function and returns a pointer to a buffer of dynamic memory whose size in bytes is determined by the **`size`** argument. The contents of the returned buffer is undetermined. Dynamic memory returned from this function is subject to the validation provided by the global variables and other tools described in this section.

Function Prototype: `ST_VOID *chk_malloc (ST_UINT size);`

Parameters:

`size`	This indicates the size in bytes of the buffer to be allocated.

Return Value:

`ST_VOID *`	<> null	This is a pointer to the allocated buffer.
	null	The memory allocation has failed.

chk_calloc

Usage: This function replaces the standard C calloc function and returns a pointer to a buffer of dynamic memory whose size in bytes is determined as a product of the **num** and **size** argument. The contents of the returned buffer are all 0x00. Dynamic memory returned from this function is subject to the validation provided by the global variables and other tools described in this section.

Function Prototype: `ST_VOID *chk_calloc (ST_UINT num, ST_UINT size);`

Parameters:

num	This indicates the number of continuous areas of memory to allocate.
size	This indicates the size in bytes of each memory are to allocate.

Return Value: ST_VOID *

<>null	This is a pointer to the allocated buffer.
null	The memory allocation failed.

chk_realloc

Usage: This function replaces the standard C realloc function and returns a pointer to a buffer of dynamic memory whose size in bytes is determined by the size argument. The contents of the returned buffer contain the contents of the old buffer. Dynamic memory returned from this function is subject to the validation provided by the global variables and other tools described in this section.

Function Prototype: `ST_VOID *chk_realloc (ST_VOID *old, ST_UINT size);`

Parameters:

`old`	This pointer indicates the old buffer.
`size`	This indicates the new size of the buffer.

Return Value:

`ST_VOID *`	<> null	This is a pointer to the resized buffer.
	null	The memory reallocation failed.

chk_free

Usage: This function deallocates a memory buffer allocated with **chk_calloc**, **chk_malloc**, or **chk_realloc**. Integrity checking is present to detect if pointers are being freed more than once, or if bogus pointers are being freed.

Function Prototype: `ST_VOID chk_free (ST_VOID *ptr);`

Parameters:

`ptr`	This is a pointer to the memory buffer that is to be deallocated.

Return Value: `ST_VOID` (ignored)

mem_chk_err

Usage: This function pointer may be set to point to an exception function in the application. The memory management tools will invoke this function pointer when a memory buffer related problem is detected.

Function Prototype: `extern ST_VOID (*mem_chk_err) (ST_VOID);`

Parameters: None

Return Value: `ST_VOID` (ignored)

10. Multi-threaded Support

This section addresses the issues related to writing multi-threaded MMS-EASE application.

NOTE: See your individual product's release notes to determine whether this functionality is supported in your environment.

general

To support multi-threaded applications in a portable manner, MMS-EASE provides a set of APIs and macros. These functions and macros are used to create, request, and release semaphore objects available in your operating system environment as well as to lock and unlock global MMS-EASE resources. Additionally, you can use the field **done_sem** in the **MMSREQ_PEND** structure to instruct MMS-EASE to signal an event semaphore when the confirm to a specific request is received. Note that if you use this feature, the user confirm callback function will not be called.

The functions and macros described below are defined in **glbsem.h**. If you use these macros, you need to define the symbol **S_MT_SUPPORT** when you compile your program.

The API makes use of the data type **ST_EVENT_SEM** used to represent a handle to an event semaphore. This data type is platform-specific. Refer to the release notes for your MMS-EASE product for more information on **ST_EVENT_SEM**.

Sample code that shows how to use the MMS-EASE multi-threaded API is available from Technical Support upon request.

Functionality

~~To enable multi-threaded support, you must call the **gs_install** function before any other APIs or macros can be used.~~

If you need to examine or modify global MMS-EASE resources (global variables or data structures), or dynamically allocated resources (**MMSREQ_PEND** or **MMSREQ_IND** structures), you must first lock these resources. When you are finished, you must unlock the resources. Use the **S_LOCK_RESOURCE** macro to gain access to global MMS-EASE resources. Use the **S_UNLOCK_RESOURCES** macro to relinquish control.

If you want to have an event semaphore signaled when the confirmation is received to a previously issued request, you need to do the following:

1. Call the **gs_get_event_sem** function to obtain a handle of type **ST_EVENT_SEM** to an event semaphore.
2. If the request is successful, call **S_LOCK_RESOURCES**.
3. Issue a MMS request.
4. Set the **done_sem** field of the **MMSREQ_PEND** structure to the handle returned from **gs_get_event_sem**.
5. Call **S_UNLOCK_RESOURCES**.
6. When the confirmation is received, MMS-EASE will signal the event semaphore using the **gs_signal_event_sem** function.
7. When the event semaphore is signaled, you can examine the **MMSREQ_PEND** to determine the outcome of the service request.
8. When you are finished with the **MMSREQ_PEND** structure, you must free it using the **ms_free_req_ctrl** function. Call **ms_clr_mvreq** first, if the request uses VMI.
9. When the event semaphore is no longer needed, you must free it using the **gs_free_event_sem**.

Multi-Threaded Functions

gs_free_event_sem

Usage: This function frees the event semaphore that was obtained using **`gs_get_event_sem`**.

Function Prototype: `ST_VOID gs_free_event_sem (ST_EVENT_SEM es);`

Parameters:

`es` This is the handle to an event semaphore that was returned from **`gs_free_event_sem`**.

Return Value: `ST_VOID` (ignored)

gs_get_event_sem

Usage: This function creates a manual-reset, non-signaled event semaphore and returns its handle to the caller.

Function Prototype: `ST_EVENT_SEM gs_get_event_sem (ST_VOID);`

Parameters: None

Return Value: `ST_EVENT_SEM` This is a handle to an event semaphore.

gs_install

Usage: This function is used to install the multi-thread API and must be called before any other API function can be used.

Function Prototype: `ST_RET gs_install (ST_VOID);`

!

Parameters: None

Return Value: `ST_RET`

SD_SUCCESS. The semaphore was installed successfully.

SD_FAILURE. Attempt to install the semaphore failed.

gs_reset_event_sem

Usage: This function is used to reset an event semaphore.

Function Prototype: `ST_VOID gs_reset_event_sem (ST_EVENT_SEM es);`

Parameters:

`es` This is the handle to an event semaphore returned from **`gs_get_event_sem`**.

Return Value: `ST_VOID` (ignored)

gs_signal_event_sem

Usage: This function is used to signal an event semaphore.

Function Prototype: `ST_VOID gs_signal_event_sem (ST_EVENT_SEM es);`

Parameters:

`es` This is the handle to an event semaphore returned from **`gs_get_event_sem`**.

Return Value: `ST_VOID` (ignored)

gs_wait_event_sem

Usage: This function is used to check the state of an event semaphore. If the state of the semaphore is signaled, the function returns immediately. Otherwise, it blocks the caller until either the semaphore is signaled or a timeout occurs.

Function Prototype:

```
ST_RET gs_wait_event_sem (ST_EVENT_SEM es,
                          ST_LONG timeout);
```

Parameters:

`es` This is the handle to an event semaphore.

`timeout` This value specifies the timeout period in milliseconds. If the timeout is 0, the function returns immediately. If the timeout is -1, the function blocks until the semaphore is signaled.

Return Value: `ST_RET`

`SD_SUCCESS`. The semaphore is signaled.

`SD_FAILURE`. The timeout period elapsed, and the semaphore is non-signaled.

1. Context Management - Introduction

MMS provides services for managing the context of communications between two MMS nodes on a network. These services are used to establish and terminate application associations and for handling protocol errors between two MMS nodes. The terms *association* and *connection* are sometimes used interchangeably. The node that initiates the association with another node is referred to as the *calling* node. The responding node is referred to as the *called* node.

In a MMS environment, two MMS applications establish an application association using the MMS Initiate service. This process of establishing an application association consists of an exchange of some parameters and a negotiation of other parameters. The exchanged parameters include information about restrictions and attributes that pertain to each node that are determined solely by that node (e.g., which MMS services are supported). The negotiable parameters, on the other hand, are either accepted by the called node or negotiated down (e.g., the maximum message size).

To summarize, the calling application issues an Initiate service request that contains information about the calling node's restrictions and attributes and a proposed set of negotiable parameters. The called node examines the negotiable parameters and adjusts them as necessary to meet its requirements and then returns the result of this negotiation and the information about its restrictions in the Initiate response. Once the calling node receives the Initiate confirmation, the application association is established and other MMS service requests can then be exchanged between the applications.

Once an application association is established either node can assume the role of client or server independent of which node was the calling or called node. For any given set of MMS services, one application assumes the client role while the other assumes the role of the server or VMD. Whether or not a particular MMS application is a client, server (VMD), or both is determined solely by the developer of the application.

MMS Associations

In a connection oriented environment such as OSI or TCP/IP[3], the MMS Initiate service is used to signal to the lower layers that a connection must be established. The Initiate service request is passed through the layers as each layer goes through its connection establishment procedure until the Initiate indication is received by the called node. The connection does not exist until after all the layers in both nodes have completed their connection establishment procedures and the calling node has received the Initiate confirmation.

In a connectionless environment, it is not strictly necessary to send the Initiate request before two nodes can communicate. In an environment where the Initiate service request is not used before other service requests are issued by a MMS client to a VMD, each application must have all the knowledge regarding the other application's exchanged and negotiated parameters using some local means (e.g., a configuration file). This foreknowledge of the other MMS applications's restrictions is the application association from the MMS perspective. Whether an Initiate service request is used or not, application associations between two MMS applications must exist before communications can take place. In some connectionless environments such as MiniMAP, MMS nodes still use the Initiate service to establish the application association before communicating.

[3] While there are profiles of OSI and TCP/IP network communications that do not require connections and can be used with MMS, most commercial MMS implementations for seven-layer LAN environments run over network stacks that require connections.

Services For Context Management

The Context Management portion of MMS contains the following services:

Initiate This service is used to exchange and negotiate the parameters required for two MMS applications to have an application association. See page 1-149 for more information on this service.

Conclude This service is used by a client to request that a previously existing application association to be terminated in a graceful manner. The conclude service allows the server to decline to terminate the association due to ongoing activities such as download sequence or file transfer. See page 1-169 for more information on this service.

Abort This service is used to terminate an application association in an ungraceful way. The server does not have the opportunity to decline an abort. Although the MMS standard does not provide any protocol for an Abort service, most MMS implementations use other network services for providing this service. See page 1-179 for more information on this service.

Cancel This service is used by a client to cancel an outstanding MMS service request (e.g., TakeControl) that has not yet been responded to by the server. See page 1-183 for more information on this service.

Reject This service is used by either the client or server to notify the other MMS application that it had received an unsupported service request or a message that was not properly encoded. See page 1-191 for more information on this service.

Common Context Management Data Structures

Below are some common data structures used in the Context Management services:

Preferred vs. Working Parameters

Preferred Initiate Parameters are variables used as default values. These variables are used to control the defaults set when **`strt_MMS`** is called. When **`strt_MMS`** is called, MMS-EASE writes these values into the corresponding members of the **`MMS_CHAN_INFO`** structure. This occurs before issuing an Initiate request using **`mv_init`**, or before responding to an Initiate indication using **`mv_init_resp`**. These preferred parameters then become the actual working parameters used. See page 1-78 for information on the **`MMS_CHAN_INFO`** structure. See page 1-149 for information on Preferred Initiate Parameters.

Presentation Context Selection

This channel-oriented array of context masks is used to allow some control over which context is to be used for sending each MMS PDU. The presentation context determines which companion standards (if any) are being supported. The entry for the channel is initialized when the connection is established as the context for the FIRST accepted context. This can be modified by the application, if desired.

When a specific request is issued (other than Initiate) over that channel, only one context must be specified to be used. This is done by modifying the appropriate member of the **`P_context_sel`** array BEFORE sending a request using a call to **`mp_xxxx`** or **`mv_xxxx`**. The presentation context selection structure is shown as follows:

```
extern ST_UINT *P_context_sel;
```

where each member of the **`P_context_sel`** array indicates the specific presentation context (contexts for an Initiate) to use for the next service request over the specified channel:

`*P_context_sel[chan]` This is a sequence of bits where each bit that is set (=1) indicates a particular presentation context or companion standard. Currently, the only available option is **`MMS_PCI 0x0001`**

AP Context Control

MMS-EASE provides a global variable that can be selected to send either IS or DIS AP Context. The default AP Context is DIS. It is suggested to stay with the default. Many implementations including MMS-EASE are set up to ignore the AP Context received. The variable is defined with ISO MMS DIS as the default:

```
extern ST_UINT mllp_ap_context_sel;

#define SM_ISO_MMS_AP_CTXT       0x01   /* ISO MMS DIS */
#define SM_MAP_MMS_AP_CTXT       0x02   /* MAP MMS DIS */
#define SM_PRIVATE_AP_CTXT       0x04   /* ISO PRIVATE */
#define SM_ISO_MMS_IS_AP_CTXT    0x08   /* ISO MMS IS  */
```

NOTE: Most DIS applications use the ISO MMS DIS AP Context, not the MAP MMS DIS AP Context.

2. Initiate Service: Establishing a Connection

This service is used to establish associations with other applications, and to agree with the attributes of these associations. This service must be completed successfully before other MMS services may be invoked.

Primitive Level Initiate Operations

There is no primitive level interface to be used with the Initiate service. The Virtual Machine Interface must be used.

Virtual Machine Initiate Operations

The following section contains information on how to use the virtual machine interface for the Initiate service. It covers available data structures used by the VMI, and the virtual machine functions that together make up this service.

- The Initiate service consists of the virtual machine functions of **`mv_init`** , **`mv_init_resp`** , **`u_init_ind`**, and **`u_mv_init_conf`**.

Virtual Machine Context Management

The virtual machine must be used for initiating associations or responding to initiate indications. The following services are performed by the virtual machine when using its context management capabilities:

1. Handles negotiations for the association control parameters used in the initiate service (**`mv_init`** or **`mv_init_resp`**), using user-supplied preferred parameters.
2. Keeps a record of the negotiated parameters and the current status of a channel on a channel by channel basis.

Context Management Variables

The following data structures are used by the MMS-EASE virtual machine to handle the context management functions associated with the Initiate service.

Preferred Initiate Parameters

The following variables describe the preferred initiate parameters that MMS-EASE writes into the corresponding members of the **`MMS_CHAN_INFO`** structure array when **`strt_MMS`** is called. After power up, the appropriate members of this structure need to be modified. This occurs before issuing an Initiate request using **`mv_init`** or before responding to an Initiate indication using **`mv_init_resp`**. The virtual machine will negotiate ”down“ by using the preferred initiate parameters as a starting point. The variables used to control the defaults set when **`strt_MMS`** is called are shown below. If you are changing these values from their defaults, you must do so before calling **`strt_MMS`**.

`extern ST_INT16 m_version = 0;` MMS DIS version number

`extern ST_INT16 m_version = 1;` MMS IS version number

extern ST_INT32 m_segsize = mms_max_msgsize;

This is the suggested maximum message size to use. Also see **file_blk_size** as documented on page 81. This preferred size parameter defaults to the value present in **mms_max_msgsize** when **strt_MMS** is called. See page 1-84 for more information. The larger this number, the more data can be transferred per MMS service request or response.

extern ST_INT16 m_maxpend_req = 5;

This is the suggested maximum number of outstanding requests to use.

extern ST_INT16 m_maxpend_ind = 5;

This is the suggested maximum number of outstanding indications to use.

extern ST_CHAR m_max_nest = 4;

This is the suggested maximum data structure nesting level to use. MMS-EASE supports a maximum of 10 levels of nesting. Each structure or array, and each structure or array within a structure or array is equivalent to one nesting level.

extern ST_UINT m_file_blk_size;

This is the suggested maximum file block size to use for file transfers. When the application does not set this variable, it defaults to 64 bytes less than **mms_max_msgsize** to allow the largest possible amount of data to be transferred per transaction.

extern ST_UINT m_download_blk_size;

This is the suggested maximum download block size to be used for downloads. When the application does not set this variable, it defaults to 64 bytes less than **mms_max_msgsize** to allow the largest possible amount of data to be transferred per transaction.

extern ST_UCHAR m_param[2] = {MPARAM0, MPARAM1};

This is the suggested parameter support to use as a requester represented as an ASN.1 bitstring, where Bit 0 is the Most Significant Bit. Each bit is set to one (1) to represent support for a particular MMS attribute, or CBB. **MPARAM0** and **MPARAM1** are constructed by ORing together the appropriate support bits to produce the desired bitstring. This is done by manipulating the **MPARAM** constants found in **mmsop_en.h**.

Parameter	Bit #	Default	Description
MPARAM_STR1	0	Set	Data of type Array is supported.
MPARAM_STR2	1	Set	Data of type Structure is supported.
MPARAM_VNAM	2	Set	Named variables are supported.
MPARAM_VALT	3	Set	Alternate Access is not supported.
MPARAM_VADR	4	Set	The address of variables is supported and can be used over the network.
MPARAM_VSCA	5	Not Set	Scattered Access is not supported[1].
MPARAM_TPY	6	Not Set	Third Party operations are not supported[1].
MPARAM_VLIS	7	Set	Variable Lists are supported.
MPARAM_REAL	8	Not Set	Floating Point is supported.
MPARAM_AKEC	9	Not Set	AcknowledgeEventCondition parameters are supported.
MPARAM_CEI	10	Not Set	Evaluation Interval parameters for monitoring are supported.

[1] Scattered Access and Third Party Operations can be supported using the PPI but not with the VMI.

Table II: Supported MMS Parameters

```
extern ST_UCHAR m_service_resp[11] = {SERV0,SERV1,SERV2,SERV3,SERV4,SERV5,
SERV6,SERV7,SERV8,SERV9,SERV10};
```

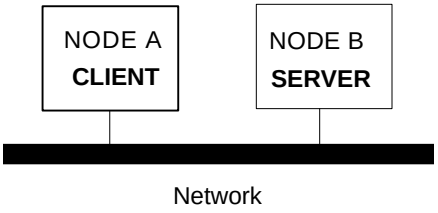
m_service_resp is an array containing an ASN.1 bitstring. It specifies which MMS services that you will be supporting. **Bit 0 is the MOST SIGNIFICANT BIT of the bitstring**. The defaults are set to enable all MMS services supported by MMS-EASE based upon the settings in the **mmsop_en.h** file. Please note that these parameters are not negotiated. The services to be supported as a responder must be specified while the remote node specifies what services to be supported as a requester. These always represent services to be supported (as a server) by the node sending the PDU. See the table shown below.

BYTE NAME	BIT	SERVICE SUPPORTED	BYTE NAME	BIT	SERVICE SUPPORTED
SERV0	0	Status	SERV5	2	Resume
	1	GetNameList		3	Reset
	2	Identify		4	Kill
	3	Rename		5	GetProgInvocAttributributes
	4	Read		6	ObtainFile
	5	Write		7	DefineEventCondition
	6	GetVariableAccessAttributes	SERV6	0	DeleteEventCondition
	7	DefineNamedVariable		1	GetEventCondAttributributes
SERV1	0	DefineScatteredAccess		2	ReportEventConditionStatus
	1	GetScatteredAccessAttributes		3	AlterEventConditionMonitoring
	2	DeleteVariableAccess		4	TriggerEvent
	3	DefineNameVariableList		5	DefineEventAction
	4	GetNamedVarListAttributes		6	DeleteEventAction
	5	DeleteNamedVariableList		7	GetEventActionAttributes
	6	DefineNamedType	SERV7	0	ReportEventActionStatus
	7	GetNamedTypeAttributes		1	DefineEventEnrollment
SERV2	0	DeleteNamedType		2	DeleteEventEnrollment
	1	Input		3	AlterEventEnrollment
	2	Output		4	ReportEventEnrollmentStatus
	3	TakeControl		5	GetEventEnrollmentAttributes
	4	RelinquishControl		6	AcknowledgeEventNotification
	5	DefineSemaphore		7	GetAlarmSummary
	6	DeleteSemaphore	SERV8	0	GetAlarmEnrollmentSummary
	7	ReportSemaphoreStatus		1	Read Journal
SERV3	0	ReportPoolSemaphoreStatus		2	WriteJournal
	1	ReportSemEntryStatus		3	InitializeJournal
	2	InitiateDownloadSequence		4	ReportJournalStatus
	3	DownloadSegment		5	CreateJournal
	4	TerminateDownloadSequence		6	DeleteJournal
	5	InitateUploadSequence		7	GetCapabilityList
	6	UploadSegment	SERV9	0	FileOpen
	7	TerminateUploadSequence		1	FileRead
SERV4	0	RequestDomainDownload		2	FileClose
	1	RequestDomainUpload		3	FileRename
	2	LoadDomainContent		4	FileDelete
	3	StoreDomainContent		5	FileDirectory
	4	DeleteDomain		6	UnsolicitedStatus
	5	GetDomainAttributes		7	InformationReport
	6	CreateProgramInvocation	SERV10	0	EventNotification
	7	DeleteProgramInvocation		1	AttachToEventCondition
SERV5	0	Start		2	AttachToSemaphore
	1	Stop		3	Conclude
				4	Cancel

Table III: MMS Supported Services

Initiate Sequence State Transitions

It is necessary to outline the sequence of events that occur within the MMS-EASE environment during association establishment. For the purposes of the scenario descriptions below, the assumption is made that the network consists of the following two nodes:



In this example, both nodes are using MMS-EASE to communicate with each other. Note that the node initiating an association is referred to as the ”calling“ node. The node responding to an association request is referred to as the ”called“ node. Once the association is established, either node can act as the client or server over that association. This depends on the negotiated parameters described previously.

Only the programmatic sequences are described here. Details such as setting up the operating system, creating the network directories, etc., are not covered in this section. Either refer to other sections of this manual, or to the documentation supplied with your network for more information.

Due to the multiple context handling features of IS, there are some functional differences in the way DIS and IS handle an Initiate. For DIS, there is a single Initiate PDU. For IS, there are multiple Initiate PDUs. There is one Initiate PDU per each proposed context. However, only one Initiate PDU can be accepted.

Following is the sequence of events that must occur to establish an association. The virtual machine functions **mv_init**, **mv_init_resp**, **u_init_ind**, and **u_mv_init_conf** are used.

1. The application programs on both Nodes A and B activate local AR Names using the **`mllp_act_ar_name`** function. This activation process enables the mapping between the AR Name and the local addressing information contained in the DIB.

    a. The programs on both Nodes A and B register these AR Names to channels using the **mllp_reg_ar_name** function. This assigns the AR Names to specific MMS-EASE channels. The channel state (**mms_chan_info[chan].ctxt.chan_state**) will be equal to **`M_IDLE`** after registration.

    b. Node B (because it is the called node which receives associate and Initiate indications from Node A) posts a listen to at least one of its registered channels using the **mllp_ass_listen** function. This allows Node B to receive associate indications on those channels. Note that because Node A only sends out requests as a calling node, Node A does not need to post any listens. The channel state (**mms_chan_info[chan].ctxt.chan_state**) will be equal to **`M_LISTEN`** on Node B after the channel is put into listen mode.

2. Node A modifies the preferred initiate parameters contained in **`MMS_CHAN_INFO`** to the desired settings. When calling Node B, it is necessary to know whether it is using IS or DIS since the Initiate PDU is different for IS and DIS. If Node B is DIS, set **`m_version = 0`**, otherwise set it equal to **1** for IS. See the section on the virtual machine context management data structures on page 1-149 for more information on the preferred initiate parameters.

3. Node A calls **`mv_init`**.

4. Internally, MMS-EASE on Node A creates an Initiate PDU using the preferred initiate parameters in **`MMS_CHAN_INFO`**. It places this PDU inside the user information field of an ACSE associate request PDU. This ACSE associate request PDU is then sent to Node B. The channel state (**`mms_chan_info[chan].ctxt.chan_state`**) is equal to **`M_ASS_REQ_PEND`** and **`M_INIT_REQ_PEND`** on Node A while waiting for the associate and initiate confirmations.
   In general, the **`mms_chan_info[chan].version`** parameter should be set to **1** (IS) for listening channels. This allows MMS-EASE to negotiate IS, if possible; otherwise, fall back to DIS. The MMS version will be set by default according to the value of the global variable **`m_version`**.

5. The ACSE associate request PDU is received by Node B and a call to the **u_mllp_a_assoc_ind** function is made. The channel state (**mms_chan_info[chan].ctxt.chan_state**) is equal to **M_ASS_IND_RCVD** on Node B at this point.

6. If **u_mllp_a_assoc_ind** returns **SD_SUCCESS**, Node B then parses the Initiate PDU, and later calls **u_init_ind** from within the **ms_comm_serve** function. The channel state (**mms_chan_info[chan].ctxt.chan_state**) remains equal to **M_ASS_IND_RCVD** on Node B.

7. If Node B returns **SD_FAILURE** from **u_mllp_a_assoc_ind**, the MMS-EASE on Node B sends a negative response to the ACSE associate request with no user information in response PDU. Channel state (**mms_chan_info[chan].ctxt.chan_state**) is equal to **N_NEG_ASS_RESP** on Node B until the negative associate response is sent, at which time, the channel state goes to **M_REPOST_LISTEN** on Node B until the channel re-enters listen mode. For negative responses only, the channel will go back automatically into the **M_LISTEN** state without the user application having to call **mllp_ass_listen** again.

8. Later **u_init_ind** is called from within the **ms_comm_serve** function. The channel state (**mms_chan_info[chan].ctxt.chan_state**) remains equal to **M_ASS_IND_RCVD** on Node B. ~~The program, within **u_init_ind** on Node B, then modifies the preferred initiate parameters.~~ It calls **mv_init_resp** if it wants to grant the association by sending a positive response, or Node B calls **mp_init_err** if it wants to refuse the association by sending a negative response. If a positive response is sent successfully, the association and channel are now considered active. They are ready for communications from Node B side. The channel state (**mms_chan_info[chan].ctxt.chan_state**) is equal to **M_INIT_URESP_WAIT** on Node B until the Initiate response is sent. It goes to **M_INIT_RESP_PEND** until the lower layers accept the associate response. At this time, the channel state goes to **M_ACTIVE** on Node B.

   If a negative response is sent, the channel state is equal to **N_NEG_ASS_RESP** on Node B until the negative associate response is accepted by the network. At this time, the channel state goes to **M_REPOST_LISTEN** until the channel re-enters listen mode, when the channel state becomes **M_LISTEN** on Node B.

9. Node A receives the ACSE Confirmation PDU, parses it, and then calls **u_mllp_a_assoc_conf** to signal to the program on Node A that an association is about to be established. The channel state is equal to **M_INIT_REQ_PEND** on Node A at this point.

10. Node A examines the negotiated ACSE parameters, and determines if these parameters are acceptable. If not, the user returns **SD_FAILURE** from **u_mllp_a_assoc_conf**. This causes MMS-EASE to send an Abort PDU to Node B. MMS-EASE then marks the outstanding Initiate request with an error, and places the request on an internal confirmation queue. When **ms_comm_serve** is called, MMS-EASE calls the **u_mv_init_conf** with a value of **ASS_USER_REJ_CONF** in the **resp_err** member of the request control data structure (of type **MMSREQ_PEND**) used for this request. Some time later, Node B receives an Abort PDU. Channel state (**mms_chan_info[chan].ctxt.chan_state**) is equal to **M_ABORT_REQ_PEND** on Node A until the abort is accepted by the lower layers of the network. Then, node A enters the **M_IDLE** state.

11. If the negotiated ACSE parameters are acceptable, the user application on Node A returns **SD_SUCCESS** from **u_mllp_a_assoc_conf**. This causes MMS-EASE to place the Initiate confirmation on an internal queue for later passing to the user using a call to the **u_mv_init_conf** function through **ms_comm_serve**. When the call to **u_mv_init_conf** is made indicating that there was no error, the association and the channel are active and normal communications can now take place. The channel state (**mms_chan_info[chan].ctxt.chan_state**) is now equal to **M_ACTIVE** on Node A.

IS vs. DIS Negotiation

For client applications calling another application, it is necessary to know whether that application is using IS or DIS since the Initiate PDU is different for IS and DIS. If the remote application is DIS, set the **m_version** to **0**; otherwise, set it to **1** (IS).

For server applications, in general, the **m_version** parameter should be set to **1** (IS) for listening channels. This allows MMS-EASE to negotiate IS if possible; otherwise, fall back to DIS.

After the MMS version has been negotiated, MMS-EASE will enforce the protocol for the selected version. This occurs in cases where there are protocol differences between IS and DIS. The decode machine will only allow the protocol that was negotiated at Initiate time.

Association Establishment

Associate Indication Information Structure (ACSE)

When an associate indication is received by MMS-EASE, the user function **u_mac_ass_ind_rcvd** is called with the specified channel, and a pointer to the structures below. The information contained in these structures can be used to determine whether to accept the association.

```
struct acse_assinfo
  {
  struct      acse_ar_info *ar_info;
  ST_UINT     ap_context;
  ST_UINT     p_contexts;
  ST_INT      num_ui;
  struct      pdu_rx_info  rxinfo[MAX_INIT_CONTEXTS];
  ST_BOOLEAN ar_matched;
  ST_RET      success;
  };
typedef struct acse_assinfo ACSE_ASSINFO;
```

Fields:

`ar_info` This is a pointer to the data structure of type **ACSE_AR_INFO** containing the ACSE level information about this association. See below for a detailed description of this structure.

`ap_context` This context mask contains the value needed to support DIS MMS as the Application Context.

`p_contexts` This contains a bit mask representing which presentation contexts (abstract syntaxes) have been requested or negotiated. The values this variable will take on correspond to the allowed values for the **p_context_sel** array. See page 1-146 for more information.

`num_ui` This indicates the number of PDUs present as user information in the ACSE Associate Indication. When MMS is the only Presentation Context (**p_context**) being negotiated there will only be one PDU. If multiple P-Contexts are being proposed, for example MMS and the MMS companion standard ROBOT, there will be more than one PDU present in the rxinfo array.

`rxinfo` /* FOR INTERNAL USE ONLY, DO NOT USE */

`ar_matched` **SD_FALSE**. The ACSE information (AP Title, AP Invoke ID, AE Invoke ID, and AE Qualifier) contained in the local **DIB** did not match that of the incoming associate request.

SD_TRUE. The ACSE information (AP Title, AP Invoke ID, AE Invoke ID, and AE Qualifier) contained in the local DIB matched that of the incoming associate request.

`success` **SD_FAILURE**. Association rejected.

SD_SUCCESS. Association accepted, confirm only.

The following structure contains two portions. The security support section of the structure is used by MMS-EASE Security Toolkit only.

```
struct acse_ar_info
  {
  ST_LONG     reserved1;
  ST_CHAR     *ar_name;
  ST_INT      transport;
  ST_UINT     AP_ctxt_mask;
  ST_INT      AP_title_form;
  ST_BOOLEAN  AP_title_pres;
  MMS_OBJ_ID  AP_title;
  ST_BOOLEAN  AP_inv_id_pres;
  ST_INT32    AP_invoke_id;
  ST_BOOLEAN  AE_qual_pres;
  ST_INT32    AE_qual;
  ST_BOOLEAN  AE_inv_id_pres;
  ST_INT32    AE_invoke_id;
  ST_INT      p_sel_len;
  ST_UCHAR    p_sel[MAX_P_SEL];
  ST_INT      s_sel_len;
  ST_UCHAR    s_sel[MAX_T_SEL];
  ST_INT      t_sel_len;
  ST_UCHAR    t_sel[MAX_T_SEL];
  ST_INT      net_addr_len;
  ST_UCHAR    net_addr[MAX_N_SEL];
  ST_CHAR     ip_addr[HOST_NAME_LEN + 1];
  ST_BOOLEAN  part_name_found;
  ST_CHAR     *part_ar_name;
  ST_UINT     part_AP_ctxt_mask;
  ST_INT      part_AP_title_form;
  ST_BOOLEAN  part_AP_title_pres;
  MMS_OBJ_ID  part_AP_title;
  ST_BOOLEAN  part_AP_inv_id_pres;
  ST_INT32    part_AP_invoke_id;
  ST_BOOLEAN  part_AE_qual_pres;
  ST_INT32    part_AE_qual;
  ST_BOOLEAN  part_AE_inv_id_pres;
  ST_INT32    part_AE_invoke_id;
  ST_INT      pci_mask;
  ST_INT      part_p_sel_len;
  ST_UCHAR    part_p_sel[MAX_P_SEL];
  ST_INT      part_s_sel_len;
  ST_UCHAR    part_s_sel[MAX_T_SEL];
  ST_INT      part_t_sel_len;
  ST_UCHAR    part_t_sel[MAX_T_SEL];
  ST_INT      part_net_addr_len;
  ST_UCHAR    part_net_addr[MAX_N_SEL];
  ST_CHAR     part_ip_addr[HOST_NAME_LEN + 1];
#ifdef SECURITY_SUPPORT
  ST_BOOLEAN  secure_assoc;
  ST_INT      algo_type;
  MMS_OBJ_ID  mech_name;
  ST_INT      auth_value_len;
  ST_CHAR     *auth_value;
  MMS_OBJ_ID  part_mech_name;
  ST_INT      part_auth_value_len;
  ST_CHAR     *part_auth_value;
  ST_INT      mech_info_len;
  ST_CHAR     *mech_info;
#endif
  };
typedef struct acse_ar_info ACSE_AR_INFO;     /* used by MMS            */
typedef struct acse_ar_info ASS_INFO;  /* used by SUIC          */
extern ASS_INFO *s_ass_info;
```

Fields:

`reserved1` **** RESERVED FOR INTERNAL USE ONLY ****

ar_name This contains the local AR Name registered on this channel.

`transport` This indicates the type of Transport to be used. The options are:

0 TP4

1 TCP/IP

AP_ctxt_mask This contains the bitmask for local AP Contexts.

AP_title_form This indicates the form that is being used for the AP Title. See **Volume 3 — Module 10 — Third Party** for more information on these two forms.

0 Form 1. The AP Title is in ASN.1 form. This form is currently not used.

1 Form 2. The AP Title is in object identifier form. This consists of members of type **MMS_OBJ_ID** and LONG integers. MMS-EASE currently defaults to this form. See **Volume 3 — Module 10 — Third Party Handling** for more information on this structure.

AP_title_pres **SD_FALSE**. The AP Title is not present.

SD_TRUE. The AP Title is included for the local application.

`AP_title` This is a structure of type **MMS_OBJ_ID** containing the AP Title (Application Process Title) registered on this channel. See **Volume 3 — Module 10 — Third Party Handling** for a detailed description of this structure. The flag **AP_title_pres** indicates whether there is a valid value for this member. This information can be optional, and will reflect whatever was in the **DIB** file.

`AP_inv_id_pres` **SD_FALSE**. The AP Invoke ID is not present.

SD_TRUE. The AP Invoke ID is included for the local application.

`AP_invoke_id` This contains the AP Invoke ID for the local application. The flag **AP_inv_id_pres** indicates whether there is a valid value in this member. This information can be optional, and will reflect whatever was in the **DIB** file.

`AE_qual_pres` **SD_TRUE**. The AE Qualifier is not present.

SD_FALSE. The AE Qualifier is included for the local application.

`AE_qual` This contains the AE Qualifier (Application Entity Qualifier) for the local application. The flag, **AE_qual_pres,** indicates whether there is a valid value in this member. This information can be optional, and will reflect whatever was in the **DIB** file.

`AE_inv_id_pres` **SD_FALSE**. The AE Invoke ID is not present.

SD_TRUE. The AE Invoke ID is included for the local application.

`AE_invoke_id` This contains the AE Invoke ID for the local application. The flag **AE_inv_id_pres** indicates whether there is a valid value in this member. This information can be optional, and will reflect whatever was in the **DIB** file.

`p_sel_len` This contains the length of the **p_sel**. **MAX_P_SEL** is set by default to be 16.

`p_sel` This contains the Presentation Selector of the local application. See page 1-40 for more information on Psels.

`s_sel_len` This contains the length of the **s_sel**. **MAX_S_SEL** is set by default to be 16.

`s_sel` This contains the Session Selector of the local application. See page 1-40 for more information on Ssels.

`t_sel_len` This contains the length of the **`t_sel`**. **`MAX_T_SEL`** is set by default to be 32.

`t_sel` This contains the Transport Selector of the local application. See page 1-40 for more information on Tsels.

`net_addr_len` This contains the length of the **`net_addr`**. **`MAX_N_SEL`** is set by default to be 20.

`net_addr` This contains the Network Address of the local application used for OSI addressing. See page 1-40 for more information on how to create a Network Address.

`ip_addr` This contains the **I**nternet **P**rotocol Address of the local application used for TCP/IP addressing. See page 1-45 for more information on the IP Address. The length of an IP Address is set by the constant **`HOST_NAME_LEN`** to be 64.

`part_name_found` **`SD_FALSE`**. The AP Title is not found in the local DIB for a partner (remote) application.

`SD_TRUE`. A partner or remote AR Name is found in the local DIB. The partner AR name matching process is as follows:

1. Only AP Titles are matched.
2. If the AP Title is not present, this flag is immediately set to FALSE.

`part_ar_name` This is a character array containing the partner or remote AR Name. It corresponds to the partner AP Title listed in the local DIB. Length of this AR Name must not be longer than 64 bytes. The extra byte is used to store the null character used by the C language.

`part_AP_ctxt_mask` This contains the bitmask for remote AP Contexts.

`part_AP_title_form` This indicates which form is being used for the partner or remote AP Title.

`0` Form 1. The partner AP Title is in ASN.1 form. This form is currently not used.

`1` Form 2. The partner AP Title is in object identifier form. This consists of members of type **`MMS_OBJ_ID`** and LONG integers. MMS-EASE currently defaults to this form. See **Volume 3 — Module 10 —Third Party Handling** for more information on this structure.

`part_AP_title_pres` **`SD_FALSE`**. The partner AP Title is not present.

`SD_TRUE`. The AP Title is included for the remote application.

`part_AP_title` = A structure of type **`MMS_OBJ_ID`** containing the partner or remote AP Title (Application Process Title). The flag **`part_ap_title_pres`** indicates whether there is a valid value for this member. See **Volume 3 — Module 10 — third Party Handling** for more information on these structures.

`part_AP_inv_id_pres` **`SD_FALSE`**. The partner AP Invoke ID is not present.

`SD_TRUE`. The AP Invoke ID is included for the remote application.

`part_AP_invoke_id` This contains the AP Invoke ID for the remote application. The flag **`AP_inv_id_pres`** indicates whether there is a valid value in this member.

`part_AE_qual_pres` **`SD_FALSE`**. The partner AE Qualifier is not present.

`SD_TRUE`. The AE Qualifier is included for the remote application.

`part_AE_qual` This contains the AE Qualifier (Application Entity Qualifier) for the remote application. The flag, **`part_AE_qual_pres,`** indicates whether there is a valid value in this member.

part_AE_inv_id_pres	**SD_FALSE**. The partner AE Invoke ID is not present. **SD_TRUE**. The AE Invoke ID is included for the remote application.
part_AE_invoke_id	This contains the AE Invoke ID for the remote application. The flag, **part_AE_inv_id_pres**, indicates whether there is a valid value in this member.
pci_mask	/* RESERVED FOR FUTURE USE, DO NOT USE */
part_p_sel_len	This contains the length of the **p_sel**. **MAX_P_SEL** is set to 16.
part_p_sel	This contains the Presentation Selector of the local application.
part_s_sel_len	This contains the length of the **s_sel**. **MAX_S_SEL** is set to 16.
part_s_sel	This contains the Session Selector of the local application.
part_t_sel_len	This contains the length of the **t_sel**. **MAX_T_SEL** is set to 32.
part_t_sel	This contains the Transport Selector of the local application.
part_net_addr_len	This contains the length of the **net_addr**. **MAX_N_SEL** is set to20.
part_net_addr	This contains the Network Address of the local application used for OSI addressing.
part_ip_addr	This null terminated string contains the **I**nternet **P**rotocol Address of the local application used for TCP/IP addressing. The length of an IP Address is set by the constant **HOST_NAME_LEN** to be 64.

NOTE: See page 1-39 for an explanation of AP Title, AP Invoke ID, AE Qualifier, and AE Invoke ID. See page 1-40 for more information on Psels, Ssels, Tsels, and Network Addressing (NSAP). See page 1-45 for more information on the IP Address.

LLP Function

The following function is used to add P-Contexts:

mllp_add_p_context

Usage: This function is used to add P-Contexts. There is a limit of sixteen P-Contexts. MMS-EASE deals with P-Contexts as a bit masked ST_UINT variable. See page 1-146 for more information. The CORE MMS context is defined as **`MMS_PCI`** (0x0001). This function is used to allow flexibility in adding and using additional contexts. It returns a context mask that can be used to employ the new context. Once a context has been added, applications can then use this context to establish connections and send MMS PDUs.

Function Prototype: `ST_UINT mllp_add_p_context (MMS_OBJ_ID *obj_id);`

Parameters:

`obj_id` This is a structure of type **`MMS_OBJ_ID`** pointing to the P-Context to be added. See **Volume 3 — Module 10 — Third Party Handling** for more information on this structure.

Return Value: `ST_UINT`

`=` null.	Error. **`mms_op_err`** is set with failure code.
`<> 0`	Returns the P-Context mask to be used for the new context that was added.

User LLP Functions

User LLP functions are pre-named, user-defined functions called by MMS-EASE when indications or confirms of the low-level LLP specific association primitives are received. These are not directly supported by MMS specifications. They consist of the functions described on the next pages.

Client Application

u_mllp_a_assoc_conf

Usage: This function is called by MMS-EASE after a positive confirmation to an Associate request is received, but before the **u_mv_init_conf** function is called. It can be used to examine the negotiated ACSE parameters (pointed to by **assinfo**) before the call to the **u_mv_init_conf** function. If the negotiated ACSE parameters are not found to be acceptable, the association can be aborted by returning a non zero value from this function. Upon returning from this function, the **u_mv_init_conf** function will be called indicating an error or acceptance of the association. This depends on the return value from this function. Refer to the section on MMS-EASE Sequence of Events starting on page 1-152 for more information on the establishment of associations.

Function Prototype:

```
extern ST_RET (*u_mllp_a_assoc_conf_fun)
                        (ST_INT chan, ACSE_ASSINFO *assinfo);
```

Parameters:

chan	This is the channel number over which the Initiate confirmation was received.
assinfo	This is a pointer to a data structure of type **ACSE_ASSINFO** containing the negotiated ACSE parameters for the association that has been established. See page 1-154 for more information on the form and content of this data structure.

Return Value:

ST_RET	**SD_SUCCESS**	ACSE Data OK. Call **u_mv_init_conf** with no response error.
	SD_FAILURE	Abort the association. Then call **u_mv_init_conf** indicating that the association was aborted.

NOTE: This function is only called if a positive confirmation to an associate request is received. If the associate confirmation is negative, only the **u_mv_init_conf** function will be called.

Data Structures Used: **ACSE_ASSINFO** **ACSE_AR_INFO** See pages 1-109 and 1-110 for detailed descriptions of these structures.

Server Application

u_mllp_a_assoc_ind

Usage: This function is called by MMS-EASE when an associate indication is received, and before the **u_init_ind** function is called. The purpose of this function is to be able to examine the ACSE information received, pointed to by **assinfo**. This is provided to determine whether to grant this association. If a zero value is returned from this function, MMS-EASE will then call **u_init_ind**. If a non-zero value is returned, MMS-EASE will send a negative response to the ACSE associate request. In the case where a negative response is sent, **u_init_ind** is not called. Refer to the section on MMS-EASE Sequence of Events starting on page 1-152 for more information on the establishment of associations.

Function Prototype:

```
extern ST_RET (*u_mllp_a_assoc_ind_fun)              (ST_INT
                          chan, ACSE_ASSINFO *assinfo);
```

Parameters:

chan	This is the channel number over which the ACSE associate indication was received.
assinfo	This is a pointer to a data structure of type **ACSE_ASSINFO** containing the ACSE parameters for the association that has been requested. See page 1-154 for more information on the form and content of this data structure.

Return Value: ST_RET

SD_SUCCESS	ACSE Data OK. **u_init_ind** will be called next.
SD_FAILURE	Error. **u_init_ind** will not be called. Instead, a negative confirmation to the associate request will be sent.

Virtual Machine Interface Functions

mv_init

Usage: This virtual machine request function allows the user to execute the Initiate service to establish an association with a remote application process over the specified channel. This function applies if using the MMS CORE context only. The values of **`version`**, **`segsize`**, **`maxpend_req`**, **`maxpend_ind`**, **`max_nest`**, **`param_supp`**, and **`service_resp`** in the **`MMS_CHAN_INFO`** structure for the specified channel are used as the preferred initiate parameters. These are sent in the Initiate PDU. See page 1-78 for more information on this structure. Refer to the section on MMS-EASE sequence of events starting on page 1-152 for more information on the establishment of associations.

Function Prototype:

```
MMSREQ_PEND *mv_init (ST_INT chan,ST_CHAR *info);
```

Parameters:

`chan`	This is the channel on which association is to be established.
`info`	This is a pointer to the AR Name (ACSE) information corresponding to the remote application process with which you are requesting that an association be established.

Return Value:

`MMSREQ_PEND *`	This is a pointer to a request control data structure of type **`MMSREQ_PEND`** that holds information regarding this request. In case of an error, the pointer is set to null and **`mms_op_err`** is written with the error code.

Corresponding Confirmation Function: **`u_mv_init_conf`**

u_init_ind

Usage: This user function is called when an Initiate indication is received. The contents and operation of this function are user-defined, and depend on the application requirements. To conform to the MMS protocol requirements, the application must do one of the following:

1) send a positive response using the virtual machine positive response function (**`mv_init_resp`**), or

2) call the appropriate primitive error response function (**`mp_init_err`**). See page 1-168 for an explanation of this function.

See page 1-58 for additional information regarding the handling of indications.

Function Prototype: `ST_VOID u_init_ind (MMSREQ_IND *ind);`

Inputs:

`ind` This pointer to the indication control structure, described on page 1-0 contains information specific to the indication PDU received.

Return Value: `ST_VOID` (ignored)

u_init_resp_done

Usage: This user function is called when an application issues a positive ACSE Associate response and the response has been sent successfully to the OSI Stack on the calling node. This function exists solely for the purpose of application notification. No special processing is required by the application in this callback function.

Function Prototype: `ST_VOID u_init_resp_done (ST_INT chan);`

Inputs:

`chan`	This is the channel over which the Associate Response was sent.

Return Value: `ST_VOID` (ignored)

mv_init_resp

Usage: This virtual machine response function allows responding to an Initiate indication if using the MMS CORE context only. An association is then established with the remote application process that has requested it. The values of **version**, **segsize**, **maxpend_req**, **maxpend_ind**, **max_nest**, **param_supp**, and **service_resp** in the **MMS_CHAN_INFO** structure for the specified channel are used as the preferred initiate parameters. MMS-EASE will use these parameters to negotiate the context, send back the response, and place the actual negotiated parameters back into **MMS_CHAN_INFO**. See page 1-78 for more information on this structure. See page 1-152 for more information on the sequence of events for establishing an association.

Function Prototype:

```
ST_RET mv_init_resp (MMSREQ_IND *ind);
```

Parameters:

`ind` This is the pointer to the indication control structure of type **MMSREQ_IND** received in the **u_init_ind** function.

Return Value: `ST_RET`

SD_SUCCESS. No Error

SD_FAILURE. Error

NOTE: If using this function and the MMS CORE context was not proposed, MMS-EASE will send a error response and will reject the connection. In this case, a listen will automatically be re-posted.

u_mv_init_conf

Usage: This virtual user confirmation function is called when a confirm to a **mv_init** is received. All virtual user confirmation functions are of the form described here. A pointer to this request control data structure is passed to this function when called. **resp_err** contains a value indicating whether an error occurred.

resp_info_ptr may contain data that the remote node sent back in the confirmation or it may contain error information if an error occurred. If no error has occurred (**req->resp_err = CNF_RESP_OK**), Initiate information is available using the **req->resp_info_ptr** pointer. Any return value is ignored.

Function Prototype: `ST_VOID u_mv_init_conf (MMSREQ_PEND *req);`

Parameters:

`req` This is a pointer to the request control structure returned from the original **mv_init** request function. See page 1-70 for more information on this structure.

For a negative response, or other errors, see **Volume 3 — Module 11 — Error Handling** for more information.

Return Value: `ST_VOID` (ignored)

NOTES:

1. Before returning from any virtual user confirmation function, **ms_clr_mvreq** must be called to clear up and free the data used by the virtual machine to handle the request and confirmation.
2. The contents of this function are completely user-defined.
3. The following documents parameter negotiation when a confirm is received:

The following parameters, described on the next page, are negotiated based on the MMS CORE context initiate if present and accepted, or the FIRST accepted P-Context (CS). It is suggest per NIST agreements that only one context is to be accepted.

Note that MMS-EASE will abort the connection if the response contains parameters that have been negotiated incorrectly.

. . .continued on the following page. . .

u_mv_init_conf . . . continued from preceding page . . .

NOTES (cont'd):

A. If the MMS CORE context is proposed and accepted, the following values are negotiated based on the Initiate confirm parameter values. These are set in **MMS_CHAN_INFO**. See page 1-78 for more information.

```
version
segsize
maxpend_req
maxpend_ind
max_nest
param_supp[2]
service_req[11]
service_resp[11]
file_blk_size
download_blk_size
```

B. If a CS P-Context is proposed and accepted (and the MMS CORE is not), the following values are negotiated based on the Initiate confirm parameter values. These are set in **MMS_CHAN_INFO**.

```
maxpend_req
maxpend_ind
max_nest
```

In this case, the parameters listed below are set to default values when connection is established. These parameters may be modified after the connection is established and before other MMS traffic begins.

`segsize`	(set = **`mms_max_msgsize`**)
`file_blk_size`	(set to fit within **`mms_max_msgsize`**)
`download_blk_size`	(set to fit within **`mms_max_msgsize`**)

The following parameters are **NOT** set at connection time. Note that they are initialized when MMS-EASE is started (**`strt_MMS`**). They should be reset at connect time. MMS-EASE does not, however, make use of these parameters internally for any primitive or virtual machine operations.

`version`	
`param_supp[2]`	(set = **`m_param_supp`**)
`service_req[11]`	(set = **`m_service_supp`**)
`service_resp[11]`	(set = **`m_service_supp`**)

mp_init_err

Usage: This primitive error response function is used to send an error response PDU in response to an Initiate indication. This should be called if you wish the Initiate request to be rejected. Fill in the **ADTNL_ERR_INFO** structure, as described under the **mp_err_resp** function before calling this function. See **Volume 3 — Module 11 — Error Handling** for more information.

Function Prototype:

```
ST_RET mp_init_err (MMSREQ_IND *ind,
                    ST_INT16 err_class,
                    ST_INT16 code);
```

Parameters:

`ind` This is a pointer to the indication control data structure of type **MMSREQ_IND** corresponding to the service request that was not performed successfully. This is the same pointer passed to the **u_init_ind** function when it was called.

`err_class` This integer contains the particular class of the error per ISO 9506. See **Volume 1 — Appendix A**. In general, class should be set equal to 8 for INITIATE PROBLEM.

`code` This integer contains the code that indicates the specific reason for the error to the Initiate service request. See **Volume 1 — Appendix A** or ISO 9506 for more information.

Return Value: `ST_RET`

SD_SUCCESS. No Error

SD_FAILURE. Error (error response not sent)

3. Conclude Service: Terminating a Connection

This service is used to conclude an established association in a graceful manner. Generally, a node requests this service indicating that all service requests have been completed and no further requests will be issued.

Primitive Level Conclude Operations

The following section contains information on how to use the paired primitive interface for the Conclude service. It covers the primitive level functions that together make up the Conclude Service.

- The Conclude service consists of the paired primitive functions of **mp_conclude**, **mp_conclude_resp**, **u_conclude_ind**, **mp_conclude_conf**, and **u_mp_conclude_done**.

Association Termination (Conclude)

It is necessary to outline the sequence of events that occur within the MMS-EASE environment during association termination. For the purposes of the following scenario description, the assumption is made that the network consists of two nodes.

In this example, both nodes are using MMS-EASE to communicate with each other. Only the programmatic sequences are described here. Details such as setting up the operating system, creating the network directories, etc., are not covered in this section. Either refer to other sections of this manual, or to the documentation supplied with your network for more information.

The sequence of events needed to establish associations was described previously. See page 1-152 for more information. The following sequence describes what happens when an association is terminated using a call to **mp_conclude**. This procedure is used to terminate gracefully an association. It involves a mutual agreement to terminate by both nodes. An association can be aborted simply by sending an Abort. In this case, it is not a graceful termination because the other node has no choice but to accept the abort, and loss of data is possible.

1. The user application on Node A calls **mp_conclude** to begin the association termination process gracefully. The conclude at this point is processed just like any other service request. The channel state (**mms_chan_info[chan]ctxt.chan_state**) is equal to **M_CONCL_REQ_PEND** on Node A.

2. On Node B, a call to **u_conclude_ind** is made when the indication is received and processed according to the scenario previously described. The channel state (**mms_chan_info[chan].ctxt.chan_state**) is equal to **M_CONCL_URESP_WAIT** on Node B.

3. The program on Node B then decides whether to grant the conclude request by calling **mp_conclude_resp** to send a positive response or **mp_conclude_err** for a negative response. You should only respond positively if you do not have any outstanding requests, indications, uploads or downloads in progress, open files, etc., and it is acceptable (to your application) to terminate the association. It is acceptable to delay responding to the Conclude indication until these conditions are met. The channel state (**mms_chan_info[chan].ctxt.chan_state**) is equal to **M_REL_IND_WAIT** on Node B when the conclude response is accepted by the lower layers of the network.

4. On Node A, the confirmation to the conclude request is received and a call to **u_mp_conclude_conf** is made. The **resp_err** member of the **MMSREQ_PEND** structure assigned to the conclude request can be examined to see if it was accepted or not. If the conclude request is not accepted by Node B, Node A can still abort the association by calling **mp_abort**.

5. If Node B responds positively to the conclude request, the MMS-EASE on Node A automatically sends an ACSE release request, as appropriate, to Node B. This is done transparently to the MMS-EASE user on Node A when a positive Conclude confirm is received. In this case, the channel state is equal to **M_REL_REQ_PEND** on Node A when the release request is accepted by the lower layers of the network.

6. Node B receives the release indication, and a call to **u_release_ind** is made. This informs the user application that a release is received and the association for this channel is about to disappear. The channel state (**mms_chan_info[chan].ctxt.chan_state**) is equal to **M_REL_IND_RCVD** on Node B at this time.
7. When the user returns from **u_release_ind** on Node B, a positive response to the release indication is sent back to Node A. A negative response to the release indication cannot be sent. The channel state is equal to **M_IDLE** on Node B when the release response is accepted by the lower layers of the network. If desired, a listen will need to be reposted.
8. When the release confirmation is received by Node A, a call is made to **u_mp_conclude_done** indicating that the association is no longer active. All the processing started by the conclude request is now complete. The channel state now is equal to **M_IDLE** on Node A.

MMS-EASE Support Functions for Association Termination

The following is a list of support functions that may be used to assist the application program in cleaning up any channel specific resources such as files, and outstanding indications. Please refer to the specific page or Module indicated for more details.

FUNCTION	DESCRIPTION
ms_clr_ind_que	Clears up indications to which have not yet been responded. See page 1-93 for more information.
ms_clr_rem_fctrl	Clears up any virtual machine data corresponding to files located on the remote node. See **Volume 3 — Module 9 — File Access and Management — Support Functions**.
ms_clr_locl_fctrl	Clears up any virtual machine data corresponding to local files. See **Volume 3 — Module 9 — File Access and Management — Support Functions**.
ms_del_domain_objs	Deletes any domain objects defined for the specified domain. This can be used to delete all the Association specific objects that have been defined for the specified channel. See page 1-78 for information on **mms_chan_info[chan].objs.aa_objs** and **Volume 2 — Module 6 — Domain Management — Support Functions**.

Paired Primitive Interface Functions

mp_conclude

Usage: This paired primitive request function sends a Conclude request PDU to terminate an association with the remote node on the specified channel gracefully. An association should be concluded when no more requests will be made over the channel. A Conclude differs from an Abort (see **mp_abort** on page 1-180) in that the remote node can deny the Conclude request, and no loss of data will occur.

Function Prototype: `MMSREQ_PEND *mp_conclude (ST_INT chan);`

Parameters:

`chan` This integer contains the channel number of the association to be terminated.

Return Value:

`MMSREQ_PEND *` This function returns a pointer to the request control data structure of type **MMSREQ_PEND** used to send the Conclude PDU. In case of an error, the pointer is set to null and **mms_op_err** is written with the error code.

Corresponding User Confirmation Function: **u_mp_conclude_conf**

Operation-Specific Data Structure Used: NONE

NOTE: When the confirmation to the conclude request is received (**u_mp_conclude_conf** is called), the association has not yet been released. MMS-EASE automatically generates an ACSE release association PDU as appropriate when the confirmation is received. When the association has been terminated by receiving a release confirmation, MMS-EASE will call the user confirmation function **u_mp_conclude_done**.

u_conclude_ind

Usage: This user function is called when a Conclude indication is received by a SERVER node. The Conclude service is used to terminate gracefully an association. To conform to the MMS protocol requirements, the application must do one of the following some time after the indication is received:

1) If it is okay to allow the association to be released, **mp_conclude_resp** should be called to conclude the association, and generate a positive response.

2) An error response (**mp_conclude_err**) can be sent if there are still outstanding requests that still need to be responded to (whether at the remote or local node). Another reason is if there is activity underway over this association that must be completed (such as a download or upload in progress) before releasing the association.

NOTE: Alternatively, responding to the Conclude indication can be delayed in some cases until such activity is completed.

Function Prototype: `ST_VOID u_conclude_ind (MMSREQ_IND *ind);`

Parameters:

`ind` This is a pointer to the indication control data structure **MMSREQ_IND** containing information specific to the Conclude indication received.

Return Value: `ST_VOID` (ignored)

Suggested Response Functions: **mp_conclude_resp**

Operation-Specific Data Structure Used: NONE

NOTE: When the association release indication is received, after sending the conclude response, MMS-EASE calls the **u_release_ind** function.

mp_conclude_resp

Usage: This primitive response function is used for sending a Conclude positive response PDU. This should be called in response to the **u_conclude_ind** function (a Conclude indication is received) provided it is valid to issue a positive Conclude response. Do not send a positive Conclude response if:

1) there are outstanding confirmed service requests at the remote node that have not yet been confirmed,

2) if the remote node has an outstanding confirmed service request locally that has not yet been responded to,

3) if there is an Upload or Download occurring, or

4) if the local node has control of a semaphore at the remote node.

In these cases, **mp_conclude_err** can be called instead to respond negatively to Conclude indication. Responding to the Conclude indication can be delayed until conditions permit a positive response. After calling this function, do not send any more responses on this channel.

Function Prototype: `ST_RET mp_conclude_resp (MMSREQ_IND *ind);`

Parameters:

`ind` This is a pointer to the indication control data structure of type **MMSREQ_IND** passed to the **u_conclude_ind** function when it was called.

Return Value: `ST_RET` **SD_SUCCESS**. No Error

SD_FAILURE. Error Code

Corresponding User Indication Function: **u_conclude_ind**

Operation-Specific Data Structure Used: NONE

NOTE: The association to be concluded is not actually released until an association release indication is received and responded to. The primitive user confirmation function **u_mp_conclude_done** is called by MMS-EASE when this occurs.

u_release_ind

Usage: This function is called by MMS-EASE when an association release indication is received. This is typically the result of a Conclude indication having been previously responded to positively. MMS-EASE will automatically send the association release response without any further action by the user's program. Do not attempt any further transfers over the specified channel as it is likely the remote node may not consider the association active anymore. This function may also be called as a result of a negative response that was sent to an Initiate request. See the section on Association Termination starting on page 1-169 for more information on terminating associations.

Function Prototype: `ST_VOID u_release_ind (ST_INT chan);`

Parameters:

`chan` This integer contains the channel number over which the association release indication was received.

Return Value: `ST_VOID` (ignored)

Operation-Specific Data Structure Used: NONE

NOTE: To determine when the association is no longer active, the channel state (**`mms_chan_info[chan].ctxt.chan_state`**) can be monitored. See page 1-78 for more information.

mp_conclude_err

Usage: This function is used to send an error response (result(-)) PDU in response to a Conclude indication (**u_conclude_ind**). This function should be called in place of the positive response function (see **mp_conclude_resp**) if you do not want to allow the Conclude request to proceed. Be sure to fill in the **ADTNL_ERR_INFO** structure before calling this function. This is described under the **mp_err_resp** function. See **Volume 3 — Module 11 — Error Handling** for more information.

Function Prototype:

```
ST_RET mp_conclude_err (MMSREQ_IND *ind,
                        ST_INT16 err_class,
                        ST_INT16 code);
```

Inputs:

`ind` This is a pointer to the indication control data structure of type **MMSREQ_IND** corresponding to the service request that was not performed successfully. This is the same pointer passed to the **u_conclude_ind** function when it was called.

`err_class` This integer contains the particular class of the error per ISO 9506. See **Volume 1 — Appendix A**. In general, class should be set equal to 9 for CONCLUDE PROBLEM.

`code` This integer contains the code indicating the specific reason for the error to the Conclude service request. See **Volume 1 — Appendix A** and ISO 9506 for more information.

Return Value: `ST_RET`

SD_SUCCESS. No Error

SD_FAILURE. Error (error response not sent)

u_mp_conclude_conf

Usage: This primitive user confirmation function is called when a VALID confirm to a **mp_conclude** request is received. The **resp_err** variable contains a value indicating whether an error occurred. **resp_info_ptr** may contain data that the remote node sent back in the confirmation, or it may contain error information if an error occurred. If no error occurs (**req->resp_err = CNF_RESP_OK**), conclude information is available to the application using the **req->resp_info_ptr** pointer. See page 1-58 for more information on the recommended handling of confirms.

Note that the function called for this event may be changed if desired; however, **u_mp_conclude_conf** is the standard default function.

Function Prototype:

```
ST_VOID u_mp_conclude_conf (MMSREQ_PEND *req);
```

Parameters:

`req` This is the pointer to the request control structure returned from the original **mp_conclude** function. See page 1-70 for more information on this structure.

For a negative response, or other errors, see **Volume 3 — Module 11 — Error Handling** for guidelines on handling errors.

Return Value: `ST_VOID` (ignored)

u_mp_conclude_done

Usage: This function is called by MMS-EASE when the conclude request, started by calling **mp_conclude**, has been completed, and the association is released (the release confirmation is received). If the return code indicates success (**code = SD_SUCCESS**), the association is no longer active. **mv_init** must be called to re-establish the association. This function may also be called when a release indication is received due to a negative confirmation to an Initiate request.

Function Prototype: `ST_VOID u_mp_conclude_done (ST_INT chan, ST_RET code);`

Parameters:

`chan` This is the number for the channel that was just released.

`code` This is the return code returned from the LLP. Zero indicates that the release was successful. If this value equals **SD_FAILURE**, it means that the release was not successful, and the association should be aborted by calling **mp_abort**.

Return Value: `ST_VOID` (ignored)

Operation-Specific Data Structure Used: NONE

4. Abort Service: Terminating a Connection Abruptly

This unconfirmed service is used to abort an established association abruptly without negotiation. A node requests this service indicating that it wants to terminate immediately all communication. The result of this abort request could be the destruction of previously issued requests and responses issued by either node.

Primitive Level Abort Operations

The following section contains information on how to use the paired primitive interface for the Abort service. It covers the primitive level functions that together make up the Abort Service.

- The Abort service consists of the paired primitive functions of **mp_abort**, **u_abort_ind**, and **u_mp_abort_done**.

Association Termination (Abort)

It is necessary to outline the sequence of events that occur within the MMS-EASE environment during an abrupt association termination. For the purposes of the following scenario description, the assumption is made that the network consists of two nodes. In this example, both nodes are using MMS-EASE to communicate with each other. Only the programmatic sequences are described here. Details such as setting up the operating system, creating the network directories, are not covered in this section. Either refer to other sections of this manual, or to the documentation supplied with your network for more information.

The other option in terminating an association deals with handling it in a non-graceful manner using the Abort service. The sequence of events for the abort is relatively simple.

1. Node A sends an abort by calling **mp_abort**. The association from the Node A side is no longer active. When the abort PDU is accepted by ACSE, MMS-EASE flags any outstanding requests with **resp_err = CNF_DISCONNECTED**. It then places these outstanding requests on its internal queue for later processing by **ms_comm_serve** in which the **u_mp_xxxx_conf** or **u_mv_xxxx_conf** functions will be called. At this point, the user may want to free up any resources by calling any of the appropriate support functions listed below. The channel state goes to **M_ABORT_REQ_PEND** at Node A until the abort is accepted by the lower layers of the network. At which time, the channel state goes to **M_IDLE**. If desired, a listen will need to be reposted.
2. Node B receives an abort indication. When the abort PDU is detected, MMS-EASE flags any outstanding requests with **resp_err = CNF_DISCONNECTED**. It then places these outstanding requests on its internal queue for later processing by **ms_comm_serve** in which the **u_mp_xxxx_conf** or **u_mv_xxxx_conf** functions will be called. The **u_abort_ind** function is then called. The channel state is now **M_IDLE** on Node B.
3. A call is made to **u_mp_abort_done** indicating that the association is no longer active. This function can be used for such processing as reposting listens, or cleaning up connection oriented resources.

Paired Primitive Interface Functions

mp_abort

Usage: This function is used for aborting an existing association on the specified channel. When an association is aborted, all outstanding requests are marked with an error code indicating that the channel was aborted. No confirmation functions will be called for outstanding requests aborted in this manner. Additionally after calling this function, you will not be able to respond to any pending indications that have not yet been responded to. **ms_clr_ind_que** should be called to free up the indications after aborting a channel. If the possible loss of data due to the unconfirmed requests that results from aborting a channel (such things as incomplete downloads and file transfers) is unacceptable, do not abort the channel; instead, **mp_conclude** should be called to terminate the association. The remote node, with which the association was established, will receive a user abort indication. Any indications or confirmations received for this channel will be rejected after calling.

Function Prototype: `ST_RET mp_abort (ST_INT chan, ST_INT reason);`

Parameters:

`chan` This is the channel number corresponding to the association to be aborted.

`reason` This value is ignored for ACSE networks because ACSE does not allow specifying a reason for an abort. It is always zero (0).

Return Value: `ST_RET`

SD_SUCCESS. The abort request was successful

`<> 0` The abort was not successful. The value of the return code is a MMS-EASE error code. See **Volume 1 — Appendix A** for a description of the MMS-EASE error codes.

Corresponding User Confirmation Function: NONE

NOTE: If the channel was in the listen mode before the abort occurred, the listen will have to be reposted. Use **mllp_ass_listen** to get an aborted channel back into listen mode.

u_abort_ind

Usage: This function is called by MMS-EASE when an association abort indication is received over the specified channel. When this function is called, the association that did exist on the specified channel is no longer active. No further transfers can occur over the channel. **`mv_init`** must be called to re-establish the association. Additionally, if your application was the ”called“ application on this channel, the listen will need to be reposted using **`mllp_ass_listen`**.

See page 1-58 for additional information on indications.

Function Prototype:

```
ST_VOID u_abort_ind (ST_INT chan,
                     ST_INT reason,          ST_BOO-
                     LEAN au_flag);
```

Parameters:

`chan` This is the number for the channel that just received an abort indication.

`reason` This value is ignored . It is always zero (0).

`au_flag` This flag indicates whether the source of the abort was the service provider or the user of the service:

`SD_FALSE`. Abort initiated by local or remote provider of the service. This is a Provider-Abort (P-ABORT).

`SD_TRUE`. Abort initiated by the remote user. This is a User-Abort (U-ABORT).

Return Value: `ST_VOID` (ignored)

Operation-Specific Data Structure Used: NONE

u_mp_abort_done

Usage: This function is called by MMS-EASE when the abort request, started by calling **`mp_abort`**, has been completed, and the channel is ready for new activity. This function can be used for such activities as reposting listens, and cleaning up connection oriented resources.

Function Prototype: `ST_VOID u_mp_abort_done (ST_INT chan);`

Parameters:

`chan` This is the number for the channel that was just aborted.

Return Value: `ST_VOID` (ignored)

Operation-Specific Data Structure Used: NONE

5. Cancel Service

This service is used to cancel a previously issued request that has not yet been completed. Only confirmed services may be canceled.

The MMS-EASE virtual machine automatically handles Cancel indications for those cases where indications have not yet been passed to the user program.

PRIMITIVE LEVEL CANCEL OPERATIONS

The following section contains information on how to use the paired primitive interface for the Cancel service. It covers the four functions that together make up the Cancel service.

- The Cancel service consists of the paired primitive functions of **`mp_cancel`**, **`u_cancel_ind`**, **`mp_cancel_resp`**, and **`u_mp_cancel_conf`**.

Paired Primitive Interface Functions

mp_cancel

Usage: This primitive request function sends a Cancel request PDU used to cancel a previously sent request. This previous request is referenced by **req**, a pointer to a request control data structure of type **MMSREQ_PEND**. This was returned from the **mp_xxxx** or **mv_xxxx** request function used to send the original request. A Cancel PDU is used to cancel outstanding requests that have not yet been responded to at the remote node. Certain virtual machine operations, such as ObtainFile and FileCopy, cannot be canceled.

Function Prototype:

```
MMSREQ_PEND *mp_cancel (MMSREQ_PEND *req);
```

Parameters:

req	This is a pointer to the request control data structure of type **MMSREQ_PEND** returned from the original **mp_xxxx** or **mv_xxxx** function when the request was sent.

Return Value:

MMSREQ_PEND *	This function returns a pointer to the same request data structure of type **MMSREQ_PEND** used to send the original request PDU. In case of an error, the pointer is set to null and **mms_op_err** is written with the error code.

Corresponding User Confirmation Function: **u_mp_cancel_conf**

Data Structure Used: **MMSREQ_PEND**

See page 1-70 for more information regarding this structure.

u_cancel_ind

Usage: This user function is called when a Cancel indication is received by a Server node, and is not handled automatically by MMS-EASE. The contents and operation of this function are user-defined, and depend on the application requirements. The application must do one of the following some time after the indication is received to conform to the MMS protocol requirements:

1) Call the primitive response function (**mp_cancel_resp**) to generate a positive Cancel or **mp_err_resp** to generate an error response to the original request, or

2) Call the virtual machine response function (**mv_cancel_resp**) to have the virtual machine respond either positively, or negatively, to both the original service request and the Cancel indication, or

3) Call the primitive negative (error) response (**mp_cancel_err**) within the user indication function.

See page 1-58 for more information on the recommended handling of indications. For cases 1 and 2 above, it up to the user application to cancel any actions that were commenced as a result of the original service indication. In fact, the MMS Specification disallows canceling unless the server can ”roll back“ any actions taken.

Function Prototype:

```
ST_VOID u_cancel_ind (MMSREQ_IND *ind);
```

Parameters:

`ind` This is a pointer to the indication control data structure of type **MMSREQ_IND** corresponding to the outstanding service request to be canceled.

Return Value: `ST_VOID`

Operation-Specific Data Structure Used: NONE

NOTE: This function can be called anytime **ms_comm_serve** is called, since most indications are not passed to the user (using a call to the appropriate **u_xxxx_ind** function) until **ms_comm_serve** is called again. This effectively provides priority to Cancel indications over other indications. Also note that if an indication has not yet been serviced (the appropriate **u_xxxx_ind** function has not yet been called), MMS-EASE will cancel it automatically. The original **u_xxxx_ind** and **u_cancel_ind** functions will not be called at all.

mp_cancel_resp

Usage: This primitive response function sends a Cancel positive response PDU to a Cancel indication. This should be called as a response to the **u_cancel_ind** function being called (received a cancel indication). It should be called after it has been determined that the requested service can be canceled. After this function is called, and the Cancel response has been sent successfully, you then MUST respond negatively to the original indication using **mp_error_resp**. If the requested operation cannot be canceled, the Cancel indication must be responded to negatively using **mp_cancel_err**. The determination of whether an indication is cancelable must be made by your application program. Certain virtual machine operations, such as ObtainFile and FileCopy, cannot be canceled. The virtual machine can also be used to handle cancels by calling **mv_cancel_resp**.

Function Prototype: `ST_RET mp_cancel_resp (MMSREQ_IND *ind);`

Parameters:

`ind` This is the pointer to the indication control data structure of type **MMSREQ_IND** corresponding to the service request (that was passed to the user using a call to a **u_xxxx_ind** function) to be canceled.

Return Value: `ST_RET` **SD_SUCCESS**. No Error

`<> 0` Error Code

Corresponding User Indication Function: **u_cancel_ind**

Operation-Specific Data Structure Used: NONE

NOTE: MMS-EASE automatically handles cancel indications for requests to cancel services that have not yet been passed to the user (using a call to a **u_xxxx_ind** function), or if there is no outstanding service request to cancel.

mp_cancel_err

Usage: This function is used to send an error response PDU (result(-)) to a Cancel indication request. This should be called if a Cancel indication is received (**u_cancel_ind** is called), and it is not possible to cancel the requested service. Either, it has already been processed and responded to, or it cannot be canceled.

Function Prototype:

```
ST_RET  mp_cancel_err   (MMSREQ_IND *ind,          ST_INT16
                           err_class,        ST_INT16 code);
```

Parameters:

`ind` This is a pointer to the indication control data structure of type **MMSREQ_IND** corresponding to the ORIGINAL service request to be canceled. This is the same pointer passed to the **u_xxxx_ind** function when the original service request was made.

`err_class` This integer contains the particular class of the error per ISO 9506. See **Volume 1 — Appendix A**. In general, set the class equal to 10 for CANCEL PROBLEM.

`code` This integer contains the specific error code indicating the reason that the Cancel could not be performed. See **Volume 1 — Appendix A** for a listing of valid MMS error codes.

Return Value: `ST_RET` **SD_SUCCESS**. No Error

`<> 0` Error (Cancel error response not sent)

NOTE: Normally MMS-EASE automatically cancels outstanding service requests if it can, or if the service request has already been responded to. In general, a negative cancel response will only need to be sent for code = OTHER or CANCEL-NOT-POSSIBLE.

u_mp_cancel_conf

Usage: This user function handles incoming cancel confirmations. This is called anytime a VALID confirm to a previously generated **mp_cancel** function is received. If no error occurs (**req->resp_err = CNF_RESP_OK**), information is available to the application using the **req->resp_info_ptr** pointer. The Cancel service is a special case because it involves the canceling of a previous service request. Please take special note of using **req** and other data described below as to whether it applies to the original service request or to the Cancel service request.

Function Prototype:

```
ST_VOID u_mp_cancel_conf (MMSREQ_PEND *req,          ST_BOO-
                          LEAN data_pres,            struct
                          err_info *info);
```

Parameters:

`req` This is a pointer to the request control data structure of type **MMSREQ_PEND** corresponding to the ORIGINAL service request that was being canceled.

`data_pres` **SD_FALSE**. The service was canceled successfully: There will be an error response to the original service request. **info** is null.

SD_TRUE. There was an error during the processing of the Cancel request at the remote node. **info** contains error information pertaining to the Cancel request.

`info` This pointer to a structure of type **ERR_INFO** contains error data if **data_pres != SD_FALSE**. This is error data concerning the Cancel service, not error data concerning the original service that was requested to be canceled. See **Volume 3 — Module 11 — Error Handling** for more information.

Return Value: `ST_VOID` (ignored)

Other Related Functions: **mp_cancel**

mp_cancel_resp

NOTES:

1. The structure pointed to by **req** is a structure of type **MMSREQ_PEND** determined at the time the original service request was made. The data contained in this structure applies to the original service, NOT to the Cancel service.
2. The memory allocated to hold the data pointed to by **info** will be de-allocated (freed up) after returning from this function.

Virtual Machine Cancel Operations

The following section contains information on how to use the virtual machine interface for the Cancel service. It covers the virtual machine response function for the Cancel service.

- The Cancel service consists of the virtual machine function of **mv_cancel_resp**.

Virtual Machine Interface Functions

mv_cancel_resp

Usage: This virtual machine function allows the user to respond to a Cancel indication (**u_cancel_ind**) without having to actually cancel the indication directly. This function, when called, will respond to both the original service request and the Cancel indication.

Function Prototype:

```
ST_RET mv_cancel_resp (MMSREQ_IND *ind,
                       ST_BOOLEAN success);
```

Parameters:

`ind` This is a pointer to an indication control data structure of the type **MMSREQ_IND** corresponding to the original service request.

`success` **SD_FALSE**. Send a positive Cancel response and an error response to the original service request.

SD_TRUE. Send a negative Cancel response and no response to the original service request.

Return Value: `ST_RET` **SD_SUCCESS**. No Error.

`<> 0` Error (Cancel response not sent).

NOTE: Indications that are being responded to by the virtual machine through a call to a **mv_xxxx_resp** function cannot be canceled.

6. Reject Service

This service is used to inform a MMS-user of the occurrence of a protocol error. These protocol errors concern PDUs that do not comply with the MMS standard.

Error responses may need to be sent to indications received for reasons such as the requested action cannot be performed, an error occurred while trying to implement the requested action, or some or all the data was invalid. To issue an error response to an indication, simply call the appropriate primitive error response function (**mp_err_resp**, **mp_cancel_err**, **mp_init_err**, or **mp_conclude_err**). However, if the MMS message (either an indication or confirm) received was ill-formed, parameters were out of range according to the MMS specification, or services were out of order, then the Reject service must be invoked. Such basic errors are called protocol errors and have nothing to do with the user's application. MMS-EASE will detect all protocol errors (that SISCO recognizes) and automatically invoke the Reject service. This causes a Reject PDU to be sent to the peer application and calling **u_reject_ind** function in the local application. However, the MMS Specification does not always make absolutely clear whether an error is a protocol error or an application error. So, in the rare case that the user application determines an error to be a protocol error (that MMS-EASE did not recognize), two functions are made available to the user application for invoking the reject service. These are: **mp_reject_ind** — for rejecting a received indication, and **mp_reject_conf** — for rejecting a received confirm. In addition, there are two other functions that are used in error handling: **u_ind_not_supp** and **u_conf_not_supp**. See explanations for these functions on the following pages.

NOTE: In general, user applications can assume that MMS-EASE will detect all protocol errors and take appropriate actions. User applications need only make calls to **mp_err_resp**, **mp_cancel_err**, **mp_init_err**, and **mp_conclude_err** to indication an application error.

Primitive Level Reject Operations

The following section contains information on how to use the paired primitive interface for the Reject service. It covers available data structures used by the PPI, and the primitive level functions that make up the Reject service.

Data Structures

Indication/Confirm

This structure contains the paired primitive information used by the **mp_reject_ind**, **mp_reject_conf**, and **u_reject_ind** functions.

```
struct reject_resp_info
  {
  ST_BOOLEAN detected_here;
  ST_BOOLEAN invoke_known;
  ST_UINT32  invoke;
  ST_INT      pdu_type;
  ST_INT16    rej_class;
  ST_INT16    rej_code;
  };
typedef struct reject_resp_info  REJECT_RESP_INFO;
```

Fields:

`detected_here` **SD_FALSE**. This indicates that a protocol error was detected at the remote application, and an incoming Reject PDU was received.

SD_TRUE. This indicates that a protocol error was detected by the local MMS-EASE application, and a Reject PDU was sent to the remote application. You will use this value for sending reject PDUs.

`invoke_known` **SD_FALSE**. This indicates that the Invoke ID of the rejected service is not known. MMS-EASE sends an Invoke ID of 0 (**invoke = 0**) in this case.

SD_TRUE. This indicates that the Invoke ID is known; use **invoke** below.

`invoke` This contains the Invoke ID, if **invoke_known != SD_FALSE**.

`pdu_type` ***** FOR INTERNAL USE ONLY, DO NOT USE *****

This is the PDU type of the service rejected. Refer to the MMS Protocol Specification for information on which MMS PDU types are supported.

`rej_class` This indicates the class of the reject error. This can be equal to the following appropriate values:

PDU-ERROR
INVALID-REQUEST
INVALID-RESPONSE
INVALID-ERROR-RESPONSE.

See ISO 9506 and **Volume 1 — Appendix A** for the values of the reject error class and their meanings.

`rej_code` This indicates the error code verifying the specific reject error for the given class (**rej_class**). See ISO 9506 and **Volume 1 — Appendix A** for the values and meanings of the various reject errors.

NOTES:

1. In those cases where Rejects occur, and

   a) the Invoke ID is known,

   b) the Invoke ID matches that of an outstanding request,

   then the user confirmation functions, **u_mp_xxxx_conf** or **u_mv_xxxx_conf** is called with **req->resp_err = CNF_REJ_ERR**. In such cases,
   points to a structure of type **REJECT_RESP_INFO**.

2. The function **u_reject_ind** is always called by MMS-EASE when a Reject occurs (either detected locally or remotely) except when the user application explicitly calls **mp_reject_ind** or **mp_reject_conf**. In fact, even if the user confirm function is called with **req->resp_err = CNF_REJ_ERR**, **u_reject_ind** is still called.

Paired Primitive Interface Functions

mp_reject_resp

Usage: This function can be used to send a Reject PDU in response to a received indication instead of sending a positive or error response. Normally, Rejects are not sent because of errors detected during processing indications. In that case, an error response (result(-)) PDU would be sent, if possible, using a call to **mp_err_resp**. MMS-EASE normally generates Reject PDUs automatically when it is appropriate to do so. However, there may be special cases that require a Reject PDU to be sent, such as a value being out of range, or excessive recursion on data structures. Refer to ISO 9506 for more information on the reasons for sending a Reject PDU versus an error response.

Function Prototype:

```
ST_RET mp_reject_resp (ST_INT chan,                    REJEC-
                                    T_RESP_INFO *info);
```

Parameters:

`chan` This is the number of the channel over which a reject indication was received.

`info` This is a pointer to an Operation-Specific data structure of type **REJECT_RESP_INFO** containing information specific to the Reject PDU to be sent.

Return Value: `ST_RET` **SD_SUCCESS**. No Error

`<> 0` Error (response not sent)

Operation Specific Data Structure Used: **REJECT_RESP_INFO**

See page 1-191 for a detailed description of this structure.

u_reject_ind

Usage: This function is called by MMS-EASE when a reject PDU is received over the specified channel, or when MMS-EASE automatically generates a Reject PDU.

Function Prototype:

```
ST_VOID u_reject_ind (ST_INT chan,
                      REJECT_RESP_INFO *info);
```

Parameters:

`ind` This is a pointer to the indication control data structure of type **`MMSREQ_IND`** passed to the **`u_xxxx_ind`** function for the indication being rejected.

`info` This is a pointer to an Operation-Specific data structure of type **`REJECT_RESP_INFO`** containing information specific to the Reject PDU that was received.

Return Value: `ST_VOID` (ignored)

Operation-Specific Data Structure Used: **`REJECT_RESP_INFO`**

See page 1-191 for a detailed description of this structure.

mp_reject_conf

Usage: This function can be used to Reject a received confirmation. Normally, incoming confirmations will not need to be rejected. MMS-EASE will normally generate Reject PDUs automatically when it is appropriate to do so. However, there may be special cases that require a Reject PDU to be sent, such as a value being out of range, or excessive recursion on data structures. Refer to ISO 9506 for more information on the reasons for rejecting a confirmation.

Function Prototype:

```
ST_RET mp_reject_conf (MMSREQ_PEND *req,
                       REJECT_RESP_INFO *info);
```

Parameters:

`req` This is a pointer to the request control data structure of type **`MMSREQ_PEND`** passed to the **`u_mp_xxxx_conf`** function for the confirmation being rejected.

`info` This is a pointer to an Operation-Specific data structure of type **`REJECT_RESP_INFO`** containing information specific to the Reject PDU to be sent.

Return Value: `ST_RET`

`SD_SUCCESS`. No Error

`<> 0` Error (response not sent)

Operation-Specific Data Structure Used: **`REJECT_RESP_INFO`**

See page 1-191 for a detailed description of this structure.

u_ind_not_supp

Usage: This function is used to fill in unused portions of the user indication function pointer table (**mms_ind_serve_fun**). It can be used to identify problems at runtime stemming from incorrect initialization or use of this table since this function should never be called during normal operation. Typically, MMS-EASE sends back a reject PDU (**unrecognized-service [class=1], confirmed request [code=1]**) if an indication is received for a service that is not supported. However, if the user indication function pointer table (**mms_ind_serve_fun**) is modified improperly, this function could get called. In writing application programs, make sure that this function is present at link time to avoid an unresolved external error.

Function Prototype:

```
ST_VOID u_ind_not_supp (MMSREQ_IND *ind);
```

Prototypes:

ind	This is a pointer to the null function pointer of type **MMSREQ_IND**.

Return Value: ST_VOID (ignored)

Operation-Specific Data Structure Used: NONE

u_conf_not_supp

Usage: This function is used to fill in unused portions of the user confirmation function pointer table (**mms_conf_serve_fun**). It can be used to identify problems at runtime stemming from incorrect initialization or use of this table since this function should never be called during normal operation. Typically, MMS-EASE sends back a reject PDU (**unrecognized-service [class=1]**, **confirmed response [code=2]**) if a confirmation is received for a service that is not supported. However, if the user confirmation function pointer table (**mms_conf_serve_fun**) is modified improperly, this function could get called. In writing application programs, make sure that this function is present at link time to avoid an unresolved external error.

Function Prototype: `ST_VOID u_conf_not_supp (MMSREQ_PEND *req);`

Parameters:

`req` This is a pointer to the null function pointer of type **MMSREQ_PEND**.

Return Value: `ST_VOID` (ignored)

Operation-Specific Data Structure Used: NONE

7. LLP Functions

The following functions are provided by MMS-EASE to perform actions necessary to interface with the **L**ower **L**ayer **P**rovider (**LLP**). These are not directly supported by the MMS specifications. There are two types of LLP functions: LLP Request functions and LLP User functions. By using these functions, it is not necessary to interact directly with the network to properly use the network services.

The following LLP Request functions are provided:

```
mllp_act_arname
mllp_act_ar_name
mllp_ass_listen
mllp_chk_req_timeout
mllp_deact_arname
mllp_deact_ar_name
mllp_init_req
mllp_init_resp
mllp_reg_ar_name
mllp_set_ae_invoke
mllp_set_ae_qual
mllp_set_ap_invoke
mllp_set_ap_title
mllp_stop_ass_listen
```

The following user LLP function is provided: `u_mllp_req_timeout`

In addition, there is a LLP error function provided: `u_llp_error_ind`

mllp_act_arname

Usage: This function is used to activate a local AR Name. It provides an additional feature over the **`mllp_act_ar_name`** function. It is used to specify the LLP type when activating an AR Name.

Once activated, this AR Name can then be registered to a channel using the **`mllp_reg_ar_name`** function.

Function Prototype:

```
ST_RET mllp_act_arname (ST_CHAR *ar_name,
                        ST_INT llp_type);
```

Parameters:

`ar_name`	This is a pointer to a character string containing the AR Name to be activated.
`llp_type`	This indicates the type of LLP to be used which is typically **`ACSE30_LLP`**.

Return Value: `ST_RET` **`SD_SUCCESS`**. AR Name activated.

`<> 0` Error Code.

NOTE: If your MMS-EASE product supports both the OSI and RFC1006 (TCP/IP) profile and both profiles are enabled, the AR Name will be activated over both OSI and RCF1006.

mllp_act_ar_name

Usage: This function is also used to activate a local AR Name. Once activated, this AR Name can then be registered to a channel using the **`mllp_reg_ar_name`** function.

Function Prototype:

```
ST_RET mllp_act_ar_name (ST_CHAR *ar_name);
```

Parameters:

`ar_name`	This is a pointer to a character string containing the AR Name to be activated.

Return Value:

`ST_RET`	**`SD_SUCCESS`**. AR Name activated.
	`<> 0` Error Code.

NOTE: If your MMS-EASE product supports both the OSI and RFC1006 (TCP/IP) profile and both profiles are enabled, the AR Name will be activated over both OSI and RCF1006.

mllp_ass_listen

Usage: This function is used to tell MMS-EASE that you want to receive associate (Initiate) indications on this channel. The channel specified must have had an AR Name activated and registered for it (see **`mllp_reg_ar_name`** and **`mllp_act_ar_name`**). If this function is not called for a given channel, only outgoing Initiate requests will be allowed for the channel.

Function Prototype: `ST_RET mllp_ass_listen (ST_INT chan);`

Parameters:

`chan` This is the number of a previously registered channel over which incoming ACSE Initiate indications are to be received.

Return Value: `ST_RET` **`SD_SUCCESS`**. Listening enabled.

`<> 0` Error Code.

NOTE: The channel state information contained in **`mms_chan_info[chan].ctxt.chan_state`** can be examined to check if listening is enabled on a given channel. See page 1-78 for more information.

mllp_chk_req_timeout

Usage: This function can be used to check for requests that have been outstanding for longer than the number of seconds specified by **`timeout`**. For each request that is older than **`timeout`** seconds, the request will be passed to the user-defined function, **`u_mllp_req_timeout`** for further processing.

Function Prototype:

```
ST_VOID mllp_chk_req_timeout (ST_INT chan,          ST_UINT32
                              timeout);
```

Inputs:

`chan`	This is the channel number over which the check for outstanding requests will be made.
`timeout`	This is the number of seconds after which a check for outstanding requests will be made.

Return Value: `ST_VOID` (ignored)

mllp_deact_arname

Usage: This function is used to de-activate a previously activated AR Name. It provides an additional feature over the original **`mllp_deact_ar_name`** function. It is used to specify the LLP type when de-activating an AR Name. Currently, this feature is not useful. However, when MINI-MAP and EPA functionality is released in MMS-EASE, then this function MUST be used to allow use of LLC.

After this function is called, MMS-EASE will not recognize the specified AR Name until it is re-activated. Any channels that have had this AR Name registered for them will be effectively made inactive and no further associations can be received or sent. Any registered ACSE channels that use this AR Name enabled for listening (see **`mllp_ass_listen`**) must first have the stop listen function (see **`mllp_stop_ass_listen`**) called to de-activate successfully. Any active associations using this AR Name must be terminated before deactivating.

NOTE: If you plan to use LLC in the future, it is suggested that this function be used instead of **`mllp_deact_ar_name.`**

Function Prototype:

```
ST_RET  mllp_deact_arname (ST_CHAR *ar_name,        ST_INT
                           llp_type);
```

Parameters:

`ar_name`	This is a pointer to a character string containing the AR Name that is to be de-activated.
`llp_type`	This indicates the type of LLP to be used. Typically, this is **`ACSE30_LLP`**.

Return Value:

`ST_RET`	**`SD_SUCCESS`**.	AR Name de-activated.
	`<> 0`	Error Code.

NOTES: The above function will not work, if:

1.If you fail to stop listening mode on any channel for which this AR Name is registered, or

2.If there is an active association on any channel using this AR Name.

mllp_deact_ar_name

Usage: This function is used to de-activate a previously activated AR Name. After this function is called, MMS-EASE will not recognize the specified AR Name until it is re-activated. Any channels that have had this AR Name registered for them will be effectively made inactive and no further associations can be received or sent. Any registered ACSE channels that use this AR Name and are enabled for listening (see **`mllp_ass_listen`**) must first have the stop listen function (see **`mllp_stop_ass_listen`**) called to de-activate successfully. Any active associations using this AR Name must be terminated before deactivating.

Function Prototype: `ST_RET mllp_deact_ar_name (ST_CHAR *ar_name);`

Parameters:

`ar_name` This is a pointer to a character string containing the AR Name to be de-activated.

Return Value: `ST_RET` **`SD_SUCCESS`**. AR Name de-activated.

`<> 0` Error Code.

NOTES: The above function will not work, if:

1. If you fail to stop listening mode on any channel for which this AR Name is registered, or
2. if there is an active association on any channel using this AR Name.

mllp_reg_ar_name

Usage: This function is used to register a previously activated AR Name to a given channel. Once an AR Name is registered, Initiate requests can then be sent for that channel. Incoming associate (Initiate) indications will be refused unless listen mode is also enabled (see **`mllp_ass_listen`**).

Function Prototype:

```
ST_RET  mllp_reg_ar_name (ST_INT chan, ST_CHAR *ar_name);
```

Parameters:

`chan`	This is the channel number over which the specified AR Name is to be registered.
`ar_name`	This is a pointer to the AR Name to be registered on the specified channel.

Return Value: `ST_RET` **`SD_SUCCESS`**. Channel registered.

`<> 0` Error Code.

mllp_set_ae_invoke

Usage: This function can be used to modify or enable/disable the AE Invoke ID for a specified AR Name. After **strt_MMS** has been called, this name will be contained in the internal ACSE DIB after the DIB file has been read. If the AE Invoke ID needs to be modified, this function should be called BEFORE activating the AR Name using a call to **mllp_act_ar_name** (in the case that the AR Name is a local one).

Function Prototype:

```
ST_RET mllp_set_ae_invoke (ST_CHAR *ar_name,
                           ST_BOOLEAN enable,
                           ST_INT32 val);
```

Parameters:

ar_name	This is a pointer to the AR Name for which the AE Invoke ID is to be modified.
enable	**SD_FALSE**. Disable the AE Invoke ID.
	SD_TRUE. Enable the AE Invoke ID.
val	This is the integer value of the AE Invoke ID to be associated with the specified AR Name. This value is a "don't care" if **enable = SD_FALSE**.

Return Value:

ST_RET	**SD_SUCCESS**.	AE Invoke ID Modified.
	<> 0	Error Code.

NOTE: This function does not apply for all operating systems. Please refer to specific release notes accompanying your software product.

mllp_set_ae_qual

Usage: This function can be used to modify the AE Qualifier for the specified AR Name contained in the internal DIB. If the AE Qualifier needs to be modified, this function should be called BEFORE activating the AR Name using a call to **`mllp_act_ar_name`** (in the case that the AR Name is a local one).

Function Prototype:

```
ST_RET mllp_set_ae_qual (ST_CHAR  *ar_name,
                         ST_BOOLEAN enable,
                         ST_INT32 val);
```

Parameters:

`ar_name` This is a pointer to the AR Name for which the AE Qualifier is to be modified.

`enable` **`SD_FALSE`**. Disable the AE Qualifier.

`SD_TRUE`. Enable the AE Qualifier.

`val` The integer value of the AE Qualifier to be associated with the specified AR Name. This value is a "don't care" if **`enable = SD_FALSE`**.

Return Value: `ST_RET` **`SD_SUCCESS`**. AE Qualifier Modified.

`<> 0` Error Code.

NOTE: This function does not apply for all operating systems. Please refer to specific release notes accompanying your software product.

mllp_set_ap_invoke

Usage: This function can be used to modify AP Invoke ID for the specified AR Name contained in the internal DIB. If the AP Invoke ID needs to be modified, this function should be called BEFORE activating the AR Name using a call to **`mllp_act_ar_name`** (in the case that the AR Name is a local one).

Function Prototype:

```
ST_RET mllp_set_ap_invoke (ST_CHAR *ar_name,
                           ST_BOOLEAN enable,
                           ST_INT32 val);
```

Parameters:

`ar_name` This is a pointer to the AR Name for which the AP Invoke ID is to be modified.

`enable` **`SD_FALSE`**. Disable the AP Invoke ID.

`SD_TRUE`. Enable the AP Invoke ID.

`val` The integer value of the AP Invoke ID to be associated with the specified AR Name. This value is a "don't care" if **`enable = SD_FALSE`**.

Return Value: `ST_RET` **`SD_SUCCESS`**. AP Invoke ID Modified.

`<> 0` Error Code.

NOTE: This function does not apply for all operating systems. Please refer to specific release notes accompanying your software product.

mllp_set_ap_title

Usage: This function can be used to modify AP Title for the specified AR Name contained in the internal DIB. If the AP Title needs to be modified, this function should be called BEFORE activating the AR Name using a call to **`mllp_act_ar_name`** (in the case that the AR Name is a local one).

Function Prototype:

```
ST_RET mllp_set_ap_title (ST_CHAR *ar_name,          ST_BOO-
                          LEAN enable,               MMS_OB-
                          J_ID *obj);
```

Parameters:

`ar_name` This is a pointer to a buffer containing the AR Name for which the AP Title is to be modified.

`enable` **`SD_FALSE`**. Disable the AP Title.

`SD_TRUE`. Enable the AP Title.

`obj` This is a pointer to the OBJECT IDENTIFIER of type **`MMS_OBJ_ID`** containing the AP Title to be associated with the specified AR Name. This input is a "don't care" if **`enable = SD_FALSE`**.

Return Value: `ST_RET` **`SD_SUCCESS`**. AP Title Modified.

`<> 0` Error Code.

NOTES:

1. Refer to page 1-39 for more information on the meaning of the AP Title.
2. This function does not apply for all operating systems. Please refer to specific release notes accompanying your software product.

mllp_stop_ass_listen

Usage: This function is used to disable listen mode on the specified ACSE channel. If Initiating over a channel that was assigned as a listening channel (using a call to **`mllp_ass_listen`**), the listen must first be stopped by calling this function. Also, if deactivating an AR Name that has been registered to a listening channel, this function must first be called to disable listening before deactivating the AR Name (using a call to **`mllp_deact_ar_name`**).

Function Prototype:

```
ST_RET  mllp_stop_ass_listen (ST_INT chan);
```

Parameters:

`chan`	The number for the channel on which to disable listening.

Return Value:

`ST_RET`	**`SD_SUCCESS`**.	Listening disabled.
	`<> 0`	Error Code.

NOTE: Examine the channel state information contained in **`mms_chan_info[chan].ctxt.chan_state`** to check if listening is enabled on a given channel. See page 1-78 for more information.

User Llp Function

u_mllp_req_timeout

Usage: This user function is called by MMS-EASE from within the **`mllp_chk_req_timeout`** function, after it has been determined that a request has been outstanding for longer than the number of seconds specified by **`timeout`**.

The contents and operation of this function are user-defined and depend on the application requirements. Some typical actions are:

Cancel the request
Abort the connection
Flag an application exception

Function Prototype:

```
ST_VOID u_mllp_req_timeout (ST_INT chan,
                            MMSREQ_PEND *req_ptr);
```

Parameters:

`chan` This is the channel number over which the LLP error condition was detected.

`req` This is a pointer to the request control data structure of type **`MMSREQ_PEND`** returned from the original **`mllp_chk_req_timeout`** function when the request was sent.

Return Value: `ST_VOID` (ignored)

User LLP Error Function

u_llp_error_ind

Usage: This function is called by MMS-EASE when an error condition in the Lower Layer Provider (LLP) has occurred. This normally occurs during association establishment or termination. Typically, the exception condition can be logged and the association for the given channel can be aborted. Depending on the type of error, you may not be able to continue communications over the given channel. See **Volume 1 — Appendix A** for a description of the various LLP error conditions that MMS-EASE detects. Refer to the section on MMS-EASE Sequence of Events starting on page 1-169 for more information on terminating associations.

Function Prototype:

```
ST_VOID u_llp_error_ind (ST_INT chan, ST_LONG code);
```

Parameters:

`chan` This is the channel number over which the LLP error condition was detected.

`code` This value indicates the type of LLP error that has occurred. There are separate error codes for ACSE. See **Volume 1 — Appendix A** for more information on the codes that can appear in this variable and their meaning.

Return Value: `ST_VOID` (ignored)

1. VMD Support Introduction

This portion of MMS-EASE provides services to manage Virtual Manufacturing Devices (VMDs).

The Virtual Manufacturing Device (VMD) Model

The primary goal of MMS was to specify a standard communications mechanism for devices and computer applications that would achieve a high level of interoperability. In order to achieve this goal, it would be necessary for MMS to define much more than just the format of the messages to be exchanged — a common message format, or protocol, is only one aspect of achieving interoperability. In addition to protocol, the MMS standard also provides definitions for:

Objects. MMS defines a set of common objects (e.g., variables, programs, events) and defines the network visible attributes of those objects (e.g., name, value, type).

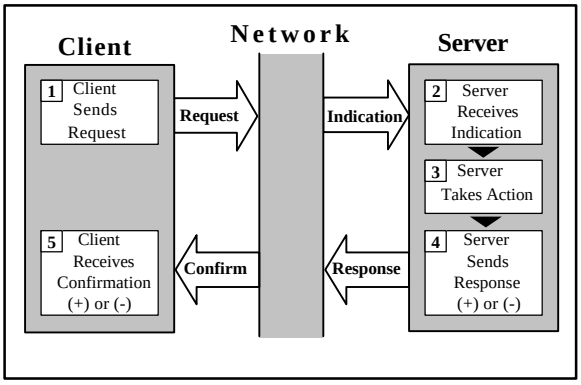
Services. MMS defines a set of communications services (e.g., read, write, delete) for accessing and managing these objects in a network environment.

Behavior. MMS defines the network visible behavior that a device should exhibit when processing these services.

This definition of objects, services, and behavior comprises a comprehensive definition of how devices and applications communicate that MMS calls the Virtual Manufacturing Device (VMD) model. The VMD model only specifies the network visible aspects of communications. The internal detail of how a real device implements the VMD model (i.e. the programming language, operating system, CPU type, input/output (I/O) systems) are not specified by MMS. By focusing only on the network visible aspects of a device, the VMD model is specific enough to provide a high level of interoperability while still being general enough to allow innovation in application/device implementation and making MMS suitable for application across a range of industries and devices types.

Client/Server Relationship

A key aspect of the VMD model is the client/server relationship between networked applications and/or devices. A server is a device or application that contains a VMD and its objects (variables, etc.). A client is a networked application (or device) that asks for data or an action from the server. In a very general sense, a client is a network entity that issues MMS service requests to a server. A server is a network entity that responds to the MMS requests of a client (see figure on right). While MMS defines the services for both clients and servers, the VMD model defines the network visible behavior of servers only.



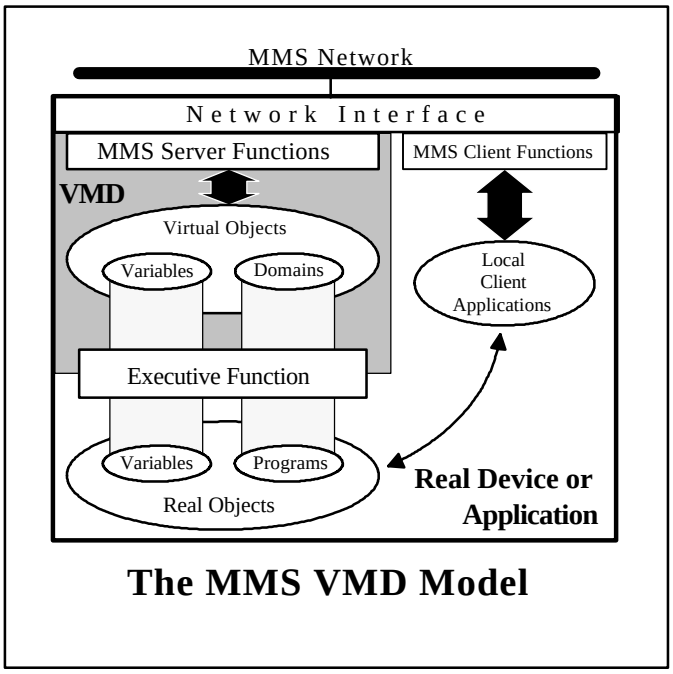
Client/Server Interactions*. MMS clients and servers interact with each other by sending/receiving request, indication, response, and confirmation Service Primitives over the network. The diagram above depicts the interactions between a client and server for a MMS Confirmed Service where 1) the client sends a request, 2) the server receives an indication, 3) the server performs the desired action, 4) the server sends a positive (+) response if the action was successful or a negative (-) response if there was an error, and 5) the client receives the confirmation (+) or (-). An Unconfirmed Service is sent by the server and has only the request and indication service primitives (see InformationReport, UnsolicitedStatus, and EventNotification for examples of unconfirmed services).*

Figure 1: Client and Server Interactions

Many MMS applications and MMS compatible devices provide both MMS client and server functions. The VMD model would only define the behavior of the server functions of those applications. Any MMS application or device that provides MMS server functions must follow the VMD model for all the network visible aspects of the server application or device. MMS clients are only required to conform to rules governing message format or construction and sequencing of messages (the protocol).

"Real" and "Virtual" Devices and Objects

There is a distinction between a real device (e.g., a PLC, CNC, or robot) and the real objects contained in it (e.g., variables, programs) and the virtual device and objects defined by the VMD model. Real devices and objects have peculiarities (a.k.a. product features) associated with them that are unique to each brand of device or application. Virtual devices and objects conform to the VMD model and are independent of brand, language, operating system, etc. Each developer of a MMS server device or MMS server application is responsible for "hiding" the details of their real devices and objects, by providing an executive function. The executive function translates the real devices and objects into the virtual ones defined by the VMD model when communicating with MMS client applications and devices. Because MMS clients always interact with the virtual device and objects defined by the VMD model, the client applications are isolated from the specifics of the real devices and objects. A properly designed MMS client application can communicate with many different brands and types of devices in the same manner because the details of the real devices and objects are hidden from the MMS client by the executive function in each VMD.



Real and Virtual Objects. *The executive function provides a translation or "mapping" between the MMS defined virtual objects and the real device objects used by the real device. Applications local to the VMD, and the objects contained in them, are only accessible to a remote MMS client application if the executive function provides the mapping function for those objects and applications. Client applications local to the VMD may access and manipulate the real objects without using MMS.*

Figure 2: The MMS VMD Model

This virtual approach to describing server behavior does not constrain the development of innovative devices and product features and improvements. The MMS VMD model places constraints only on the network visible aspects of the virtual devices and objects, not the real ones.

MMS Device and Object Modeling

The implementor of the executive function (the application or device developer) must decide how to "model" the real objects as virtual objects. The manner in which these objects are modeled is critical to achieving interoperability between clients and servers among many different developers. Inappropriate or incorrect modeling can lead to an implementation that is difficult to use or with which to interoperate.

For instance, take the situation of a PLC that contains a ladder program (a real object). The MMS implementor (the designer of the executive function) wishes to allow external client applications to copy the program from the PLC to another computer. For the purposes of this example we shall assume that the MMS VMD model gives the implementor the choice of modeling the ladder program object as a variable or domain object (both virtual objects). The choice of which virtual object to map to the real ladder program object is critical because MMS provides a set of services to manipulate variables that are quite different from the services used to

manipulate domains. Variables can be read individually or in a list of typed data. Domains can be copied in whole only. If the ladder program is modeled as a MMS variable it makes the task of performing a simple copying of the program complex because the nature of the ladder program data (typically a large block of contiguous memory) would result in an extremely large variable that would be difficult to access easily. If the ladder program is modeled as a domain, there are specific MMS services provided for uploading and downloading the large blocks of untyped data typical of ladder programs. An incorrect modeling choice can make the real object difficult to access.

In some cases it makes sense to represent the same real object with two different MMS objects. For instance, a large block of variables may also be modeled as a domain. This would provide the MMS client the choice of services to use when accessing the data. The variable services would give access to the individual elements in the block. The domain services would allow the entire block to be read/uploaded or written/downloaded as an element of a program invocation.

MMS Objects

MMS defines a variety of objects that are found in many typical devices and applications requiring real-time communications. A list of these objects is given below. For each object there are corresponding MMS services that let client applications access and manipulate those objects. The model for these objects and the services available are described in more detail later.

- **VMD**. The device itself is an object.
- **Domain**. Represents a resource (e.g., a program) within the VMD.
- **Program Invocation**. A runnable program consisting of one or more domains.
- **Variable**. An element of typed data (e.g., integer, floating point, array).
- **Type**. A description of the format of a variable's data.
- **Named Variable List**. A list of variables that is named as a list.
- **Semaphore**. An object used to control access to a shared resource.
- **Operator Station**. A display and keyboard for use by an operator.
- **Event Condition**. An object that represents the state of an event.
- **Event Action**. Represents the action taken when an event condition changes state.
- **Event Enrollment**. Which network application to notify when an event condition changes state.
- **Journal**. A time based record of events and variables.
- **File**. A file in a filestore or fileserver.
- **Transaction**. Represents an individual MMS service request. Not a named object.

Object Attributes and Scope

Associated with each object are a set of attributes that describe that object. MMS objects have a name attribute and other attributes that vary from object to object. Variables have attributes such as name, value, and type while other objects, program invocations for instance, have attributes such as name and current state.

Subordinate objects exist only within the scope of another object. For instance, all other objects are subordinate to, or contained within, the VMD itself. Some objects, such as the operator station object for example, may be subordinate only to the VMD. Some objects by be contained within other objects such as variables contained within a domain. This attribute of an object is called its *scope*. The object's scope also reflects the lifetime of an object. An object's scope may be defined to be:

- **VMD-Specific**. The object has meaning and exists across the entire VMD (is subordinate to the VMD). The object exists as long as the VMD exists.
- **Domain-Specific**. The object is defined to be subordinate to a particular domain. The object will exist only as long as the domain exists.

- **Application-Association-Specific**. Also referred to as AA-Specific. The object is defined by the client over a specific application association and can only be used by that specific client. The object exists as long as the association between the client and server exists.

The name of a MMS object must also reflect the scope of the object. For instance, the object name for a domain-specific variable must not only specify the name of the variable within that domain but also the name of the domain. Names of a given scope must be unique. For instance, the name of a variable specific to a given domain must be unique for all domain specific variables in that domain. Some objects, such as variables, are capable of being defined with any of the scopes described above. Others, like semaphores for example, cannot be defined to be AA-specific. Still others, like operator stations for example, are only defined as VMD-specific. When an object like a domain is deleted, all the objects subordinate to that domain must also be deleted.

The VMD Object

The VMD itself is also an object and has attributes associated with it. Some of the network visible attributes for a VMD are:

Capabilities — A capability of a VMD is a resource or capacity defined by the real device. There can be more than one capability to a VMD. The capabilities are represented by a sequence of character strings. The capabilities are defined by the implementor of the VMD and provides useful information about the real device or application.

Logical Status — Logical status refers to the status of the MMS communication system for the VMD which can be: STATE-CHANGES-ALLOWED, NO-STATE-CHANGES-ALLOWED or ONLY-SUPPORT-SERVICES-ALLOWED.

Physical Status — Physical status refers to the status of all the capabilities taken as a whole which can be equal to: OPERATIONAL, PARTIALLY-OPERATIONAL, INOPERABLE or NEEDS-COMMISSIONING.

VMD Support Services

There are six services related to VMD support:

Status — This confirmed service is used by a client to obtain the logical and physical status of the VMD. The Status and UnsolicitedStatus services also supports access to implementation-specific status information (called *local detail*) defined by the implementor of the VMD. See page 2-15 for more information.

UnsolicitedStatus — This unconfirmed service is used by a server (VMD) to report its status to a client unsolicited by the client. See page 2-21 for more information.

GetNameList — This confirmed service allows a client to obtain a list of named objects defined within the VMD. See page 2-25 for more information.

Identify — This confirmed service allows the client to obtain information about the MMS implementation such as the vendor's name, model number, and revision level. See page 2-33 for more information.

Rename — This service is used to rename a MMS object at the server. See page 2-41 for more information.

GetCapabilityList — This service is used to obtain a list of capabilities of a VMD. See page 2-47 for more information.

The MMS-EASE virtual machine defines several data structures and functions. These assist applications with managing information relating to MMS VMDs. MMS-EASE supports the ability to model multiple VMDs within a single application process. For each VMD, it maintains an alphabetized list of named domains, and other MMS objects. These can be manipulated using the appropriate support functions such as **`ms_add_named_dom`**, and **`ms_del_named_dom`**.

VMD Management

MMS-EASE implements a sophisticated system for managing multiple VMDs within the same environment. The MMS-EASE system allows the creation and deletion of VMDs using the appropriate support functions (**`ms_create_vmd`**, **`ms_dismantle_vmd`**, and **`ms_delete_vmd`**). When a VMD is created, you are, in effect, telling MMS-EASE to allocate the memory to hold the VMD control structure. This is described on the following pages. MMS-EASE then keeps such MMS objects as domains, variables, types added into this VMD separate from other VMDs that are defined. In particular, information contained in one VMD cannot be shared by another VMD. For example, the same types will have to be defined in two different VMDs if they are needed in both VMDs. MMS-EASE treats each VMD defined as its own operating environment. It should be pointed out that most applications will only need to use one VMD. In this case, the user application need not explicitly create a VMD, since one is created within **`strt_MMS`** by default.

MMS-EASE also allows deleting VMDs in two ways. A VMD can be **dismantled**. This means that not only is the VMD deleted, but also the entire contents of the VMD including the MMS objects such as program invocations, domains, and variables. A VMD can be **deleted** if it is already an empty shell (not containing any objects). A good way to reset a VMD to an initial, empty state is to dismantle it, and then, recreate it.

2. Virtual Machine Interface

Once all the VMDs, and their corresponding domains, types, variables are defined, only one VMD must be selected to be used at a time. When a virtual machine response function is called, select a VMD to use as a server. When a MMS-EASE support function or virtual machine request function is called, you can select the same VMD or another VMD. MMS-EASE allows independent selections as a server versus client, or on a service by service basis, according to the application's needs.

To select which VMD is to be used as a Client, simply write the **m_vmd_select** variable with a pointer to the specific VMD control structure (**VMD_CTRL**) to be used. See page 2-10 for more information on this variable. Once this variable is written with a valid pointer to an existing VMD, all further calls to virtual request (**mv_xxxx**) or support (**ms_xxxx**) functions use that VMD to find any required information it needs (such as domain, variable, type). All the information contained in any other VMD is ignored until **m_vmd_select** is changed to point to another VMD. The complete list of VMDs is always referenced from the head of the list using the pointer **m_vmd_ctrl_list**.

To select a VMD to use as a Server, a pointer should be written to the specific VMD to be supported as a server into the **objs.vmd** member of the **MMS_CHAN_INFO** structure corresponding to the channel to which the VMD is assigned. This potentially allows the assignment of a different VMD to each active channel. Once the **objs.vmd** member has a pointer to a valid VMD control structure, all virtual response functions (**mv_xxxx_resp**), called for indications received on that channel, will use the **objs.vmd** to find out from which VMD to obtain the required domain, variable, or type information. This allows a local MMS-EASE application to act as a server for more than one VMD. The mapping of each logical VMD for which MMS-EASE allocates memory to an actual physical device is the responsibility of your application program. See **Volume 1 — Module 2 — Channel Information Structure** for more information on the **MMS_CHAN_INFO** structure.

However, it may not be necessary to support multiple VMDs based upon your application program's requirements. In this case, simply ignore all the VMD management functions supplied by MMS-EASE. When MMS-EASE first powers up, it allocates the memory for the first VMD, and writes a pointer to this VMD to **m_vmd_select**, **m_vmd_ctrl_list**, and the **objs.vmd** members of all the allocated channels. This means that if multiple VMDs are not supported, it is assumed that **m_vmd_ctrl_list** points to a list of one VMD, and that **m_vmd_select** and the **objs.vmd** members point to that VMD. The rest of your application program will never need to manipulate **m_vmd_select** or **objs.vmd**. Please refer to page 10 for information on how MMS-EASE uses the VMD control structure.

NOTES:

1. When using the **m_vmd_select** variable to select among different VMDs for use during the processing of a virtual read request (**mv_read** or **mv_readvars**), make sure that the **m_vmd_select** pointer is not changed until after confirmation to the request has been received using a call to **u_mv_read_conf** or **u_mv_read_vars_conf**. The Virtual Machine needs to reference type information possibly from a VMD. Meanwhile, should that VMD be changed in the interim period of the request being issued, and the response being received, a different VMD would be referenced for this type information. It might be possible for that type information to either not exist, or be different from the original VMD. See page 2-153 for more information on the read service. It is recommend that a single VMD control structure be assigned to hold all client VMD information. This will simplify your program by separating the client VMDs from the server VMDs.
2. Do not delete the first VMD that MMS-EASE allocates upon power up without creating a second VMD to take its place. There must be at least one VMD defined for the virtual machine to function properly.

MMS-EASE Object Database

The diagram, on the following page, shows how the **MMS_CHAN_INFO** and **VMD_CTRL** structures relate to other important MMS-EASE data structures when using the database. Arrows between the data structures are meant to show the linkage reference points in the database when adding MMS-EASE objects.

There are three variables that can be set before adding any NamedVariables, NamedVariableLists, or Domains to the database (e.g., before calling **ms_add_named_var**, **ms_add_nvlist**, or **ms_add_named_domain**).

1. Set the global variable **max_mmsease_vars** to the number of NamedVariables to be allowed in each domain or VMD. This is different from the default MMS-EASE behavior that uses this variable as the total for the entire application. If this variable is not set, a default value will be used.

   Example: `max_mmsease_vars = 50000;`

2. Set the global variable **max_mmsease_nvlists** to the number of NamedVariableLists to be allowed in each domain or VMD. This is different from the default MMS-EASE behavior that uses this variable as the total for the entire application. If this variable is not set, a default value will be used.

   Example: `max_mmsease_nvlists = 1000;`

3. Set the global variable **max_mmsease_doms** to the number of NamedDomains to be allowed in the application. If this variable is not set, a default value of 20 will be used.

   Example: `max_mmsease_doms = 1000;`

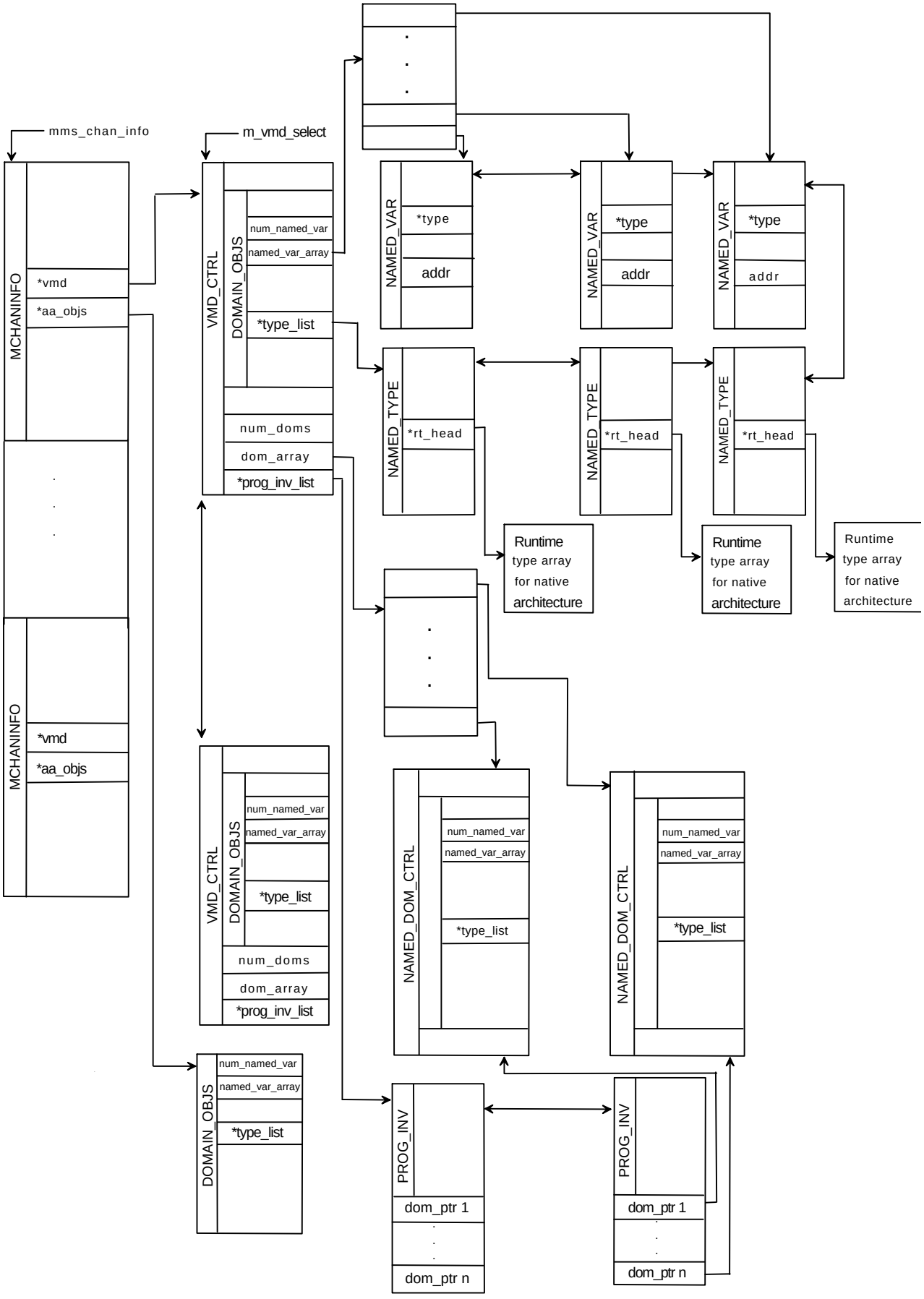


Figure 3: MMS-EASE Object Database Layout

VMD Control Data Structures

The following data structures are used by the MMS-EASE VMD management system. There is one structure of type **VMD_CTRL** for each VMD defined.

```
struct vmd_ctrl
  {
  DBL_LNK          link;
  DOMAIN_OBJS      vmd_wide;
  NAMED_DOM_CTRL   *dom_list;
  struct prog_inv  *prog_inv_list;
  struct oper_stat *op_stat_list;
  struct ae_ctrl   *ae_list;
  ST_CHAR          *user_info;
/* The following elements used instead of dom_list if binary search */
/* used instead of linked lists.                                    */
  ST_INT           max_doms;
  ST_INT           num_doms;
  NAMED_DOM_CTRL   **dom_array;
  };
typedef struct vmd_ctrl   VMD_CTRL;
```

Fields:

`link` /* FOR INTERNAL USE ONLY, DO NOT USE */

`vmd_wide` This structure of type **DOMAIN_OBJS** contains the list of VMD-specific objects corresponding to this VMD.

`dom_list` This pointer to a structure of type **NAMED_DOM_CTRL** points to the beginning of the list of named domains defined for this VMD. This element is used for linked lists.

`prog_inv_list` This pointer to a structure of type **prog_inv** points to the beginning of the list of program invocations defined for this VMD.

`op_stat_list` This pointer to a structure of type **oper_stat** points to the beginning of the list of operator stations defined for this VMD.

`ae_list` This pointer to a structure of type **ae_ctrl** points to the beginning of the list of AE Invocations defined for this VMD. This member is reserved for use with MMSI.

`user_info` This is the pointer to some user specific information that can be tagged to this VMD. The contents and meaning of the data pointed to by **user_info** is completely user-defined. MMS-EASE does not use this pointer.

The following elements used instead of **dom_list** if binary search used instead of linked lists.

`max_doms` This is the maximum number of domains to be included in a binary search. This number may be set by the application before adding the first Named Domain or it will default to the MMS-EASE global value **max_mmsease_doms** at the time the first Named Domain is added to the VMD.

`num_doms` This is the current number of Named Domains associated with the VMD and is used to access the valid pointers in the **dom_array**.

`dom_array` This array of pointers points to the **NAMED_DOM_CTRL** structures associated with the VMD.

```
extern VMD_CTRL *m_vmd_ctrl_list;
```

This variable is a pointer to the first VMD control structure of type **VMD_CTRL** that begins the list of defined VMDs. This variable is written at power up, and should not be overwritten by your program. Do not delete the VMD to which this variable points.

```
extern VMD_CTRL *m_vmd_select;
```

This variable is used to select which VMD to use as the current VMD when manipulating a VMD locally (as a client). It must be set to point to a defined VMD before calling any virtual request or support function.

VMD Support Functions

These functions are called by the user's application program to update the VMD database.

ms_create_vmd

Usage: This support function creates the new VMD control structures needed to add a new VMD into the MMS-EASE database. This new VMD is then added to the Virtual Machine Database. The information passed to this function is used to determine the channel(s) to which this VMD should be assigned.

Function Prototype: `VMD_CTRL *ms_create_vmd (ST_INT num, ST_INT *arr);`

Parameters:

`num` This contains the number of elements (channels) in the array pointed to by **`arr`**. When **`num = 0`**, the VMD is created and attached to the **`VMD_CTRL`** list but not assigned to any channel.

`arr` This pointer to the beginning of an array of integers contains a list of all the channel numbers to which this VMD will be assigned. This is the address of the first element in the array (**`&arr[n]`**). MMS-EASE writes a pointer to this newly created VMD into the **`mms_chan_info[chan].objs.vmd`** member corresponding to each of the channel numbers specified in the array.

Return Value:

`VMD_CTRL *` This pointer to the VMD control structure for the newly created VMD. In case of an error, the pointer is set to null and **`mms_op_err`** is written with the error code.

Data Structures Used:

`mms_chan_info.obj.vmd` See **Volume 1 — Module 2 — Channel Information Structure** for information on this structure.

`VMD_CTRL` See page 2-10 for more information.

ms_delete_vmd

Usage: This support function deletes an empty **VMD_CTRL** structure from the MMS-EASE database. This means the VMD cannot contain any objects such as domains, or types. If the VMD does contain any objects, it will NOT be removed from the database. The support function, **ms_dismantle_vmd**, described on the next page, can delete a VMD containing objects.

Function Prototype: `ST_RET ms_delete_vmd (VMD_CTRL *vmd);`

Parameters:

vmd	This pointer of structure type **VMD_CTRL** points to the empty VMD to be deleted.

Return Value: ST_RET

SD_SUCCESS. No error encountered in deleting the specified VMD.

SD_FAILURE. Error encountered in deleting the specified VMD. The most probable cause is that the VMD contained objects.

Data Structures Used:

VMD_CTRL	See page 2-10 for more information.

ms_dismantle_vmd

Usage: This support function deletes a VMD from the MMS-EASE database. In addition, it automatically deletes the contents of all VMD objects such as program invocations, domains, or types. To prevent undefined references, the application should make sure that none of the **mms_chan_info[chan].objs.vmd** members or the **m_vmd_select** pointer is pointing to this VMD before deleting it. See page 2-5 for more information on the **m_vmd_select** pointer.

Function Prototype: `ST_VOID ms_dismantle_vmd (VMD_CTRL *vmd);`

Parameters:

vmd	This pointer of structure type **VMD_CTRL** points to the VMD and its associated objects to be deleted.

Return Value: ST_VOID (ignored)

Data Structures Used:

VMD_CTRL	See page 2-10 for more information.
MMS_CHAN_INFO	See **Volume 1 — Module 2 — Channel Information Structure** for more information.

NOTE: If executable size is a consideration, the application must keep track of the contents of the VMD. The individual objects should be dismantled by making calls to the **ms**_ support functions directly (such as **ms_del_domain_objs**, **ms_del_all_named_doms**, **ms_del_all_named_pis**, **ms_del_all_named_types**, and finally **ms_delete_vmd**). In memory critical applications, the use of **ms_dismantle_vmd** is only recommended when there are many different types of objects in the VMD. The **ms_dismantle_vmd** function will delete all MMS-EASE supported objects present in the VMD by automatically calling all these functions. Executable size increases somewhat due to all the MMS-EASE object removal functions that automatically get linked into the executable.

3. Status Service

This service is used to allow a client device to determine the general condition or status of a server node.

Primitive Level Status Operations

The following section contains information on how to use the paired primitive interface for the Status service. It covers available data structures used by the PPI, and the four primitive level functions that together make up the Status service. See **Volume 1 — Module 2 — General Sequence of Events — Confirmed Services** for a detailed explanation of the sequence of events that occur within the MMS-EASE environment during confirmed services.

- The Status service consists of the paired primitive functions of **`mp_status`**, **`u_status_ind`**, **`mp_status_resp`**, and **`u_mp_status_conf`**.

Data Structures

Request/Indication

This operation specific structure described below is used by the client to issue the Status request (**`mp_status`**). It is received by the server when a Status indication (**`u_status_ind`**) is received.

```
struct status_req_info
  {
  ST_BOOLEAN extended;
  };
typedef struct status_req_info STATUS_REQ_INFO;
```

Fields:

`extended` **`SD_FALSE`**. Response should be generated using the non-extended derivation method.

`SD_TRUE`. Response should be generated using an extended derivation method if available (such as invoking a self diagnostics routine).

Response/Confirm

This operation specific data structure described below is used by the server in issuing the Status response (**`mp_status_resp`**). It is received by the client when a Status confirm (**`u_mp_status_conf`**) is received.

```
struct status_resp_info
  {
  ST_INT16    logical_stat;
  ST_INT16    physical_stat;
  ST_BOOLEAN local_detail_pres;
  ST_INT     local_detail_len;
  ST_UCHAR   local_detail[MAX_STAT_DTL_LEN];
  };
typedef struct status_resp_info STATUS_RESP_INFO;
```

Fields:

`logical_stat` This required field indicates the logical status of the VMD:

- **0** State changes are allowed (All supported services can be used).
- **1** No state changes are allowed (services that modify the state of a VMD object).
- **2** Limited services are permitted (Abort, Conclude, Status, and Identify)
- **3** Support services are allowed (all services supported by the VMD except Start, Stop, Reset, Resume and Kill).

`physical_stat` This required field indicates the physical status of the VMD:

- **0** Fully operational.
- **1** Partially operational.
- **2** Inoperable.
- **3** Needs Commissioning (manual intervention may be needed).

`local_detail_pres` **SD_FALSE**. Do Not include **local_detail** in PDU.

SD_TRUE. Include **local_detail** in PDU.

`local_detail_len` This is the length, IN BITS, of the **local_detail**. This cannot be greater than 128 bits in length. **MAX_STAT_DTL_LEN** is set by default to be 16 bits.

`local_detail` This implementation-specific bitstring contains additional data about the status of the VMD. It is defined by the particular VMD.

Paired Primitive Interface Functions

mp_status

Usage: This primitive request function sends a Status request PDU. It uses the data from a structure of type **STATUS_REQ_INFO**, pointed to by **info**. This function is used to obtain the status of a remote node.

Function Prototype:

```
MMSREQ_PEND *mp_status (ST_INT chan,                STA-
                                    TUS_REQ_INFO *info);
```

Parameters:

chan	This integer contains the channel number over which the Status PDU is to be sent.
info	This pointer to an Operation Specific data structure of type **STATUS_REQ_INFO** contains information specific to the Status PDU to be sent.

Return Value:

MMSREQ_PEND *	This pointer to the request control data structure of type **MMSREQ_PEND** is used to send the Status PDU. In case of an error, the pointer is set to null and **mms_op_err** is written with the error code.

Corresponding User Confirmation Function: **u_mp_status_conf**

Operation-Specific Data Structure Used: **STATUS_REQ_INFO**

See page 2-15 for a detailed description of this structure.

u_status_ind

Usage: This user function is called when a Status indication is received by a server node. The contents and operation of this function are user-defined, and depend on the application requirements. To conform to the MMS protocol requirement, the application must do one of the following some time after the indication is received:

1) call the positive response using the primitive response function (**`mp_status_resp`**) after obtaining the Status from the remote node, or

2) call the appropriate primitive negative (error) response function (**`mp_err_resp`**). See **Volume 3 — Module 11 — MMS-EASE Error Handling** for more information on this function.

See **Volume 1 — Module 1 — Function Classes — User Indication Function Class** for additional information regarding the handling of indications.

Function Prototype:

```
ST_VOID u_status_ind (MMSREQ_IND *ind);
```

Parameters:

`ind` This pointer to the **`MMSREQ_IND`** structure is described in **Volume 1 — Module 2 — Indication and Request Control Data Structures**. This pointer must be used to send the response to this indication. The primary item of interest in this structure is **`ind->resp_info_ptr`**. This is a pointer to the operation specific data structure for a Status indication (**`STATUS_REQ_INFO`**). This pointer will always be valid when **`u_status_ind`** is called.

Return Value: `ST_VOID` (ignored)

Operation-Specific Data Structures Used: **`STATUS_REQ_INFO`**

See page 2-15 for a detailed description of this structure.

mp_status_resp

Usage: This primitive response function sends a positive Status response PDU using the data from a structure of type **STATUS_RESP_INFO**, pointed to by **info**. It should be called after the **u_status_ind** function is called (a Status indication is received), and after the status has been successfully determined.

Function Prototype:

```
ST_RET mp_status_resp (MMSREQ_IND *ind,
                       STATUS_RESP_INFO *info);
```

Parameters:

`ind` This pointer to the indication control structure of type **MMSREQ_IND** is received in the indication service function, **u_status_ind**.

`info` This pointer to an Operation Specific data structure of type **STATUS_RESP_INFO** contains information specific to the Status response PDU to be sent.

Return Value: `ST_RET` **SD_SUCCESS**. No Error.

`<> 0` Error Code.

Corresponding User Indication Function: **u_status_ind**

Operation-Specific Data Structure Used: **STATUS_RESP_INFO**

See page 2-15 for a detailed description of this structure.

u_mp_status_conf

Usage: This primitive user confirmation function is called when a valid Status confirm to a **mp_status** is received. **resp_err** contains a value indicating whether an error occurred. **resp_info_ptr** may contain data that the remote node sent back in the confirm, or it may contain error information if an error occurred. If no error has occurred (**req->resp_err = CNF_RESP_OK**), Status information is available to the application using the **req->resp_info_ptr** pointer. See **Volume 1 — Module 1 — Function Classes — Primitive User Confirmation Functions Class** for more information on the recommended handling of confirms.

Note that the function called for this event may be changed if desired; however, **u_mp_status_conf** is the standard default function.

Function Prototype: `ST_VOID u_mp_status_conf (MMSREQ_PEND *req);`

Parameters:

`req` This pointer to the **MMSREQ_PEND** structure is returned from the original **mp_status** request function. See **Volume 1 — Module 2 — Indication and Request Control Data Structures** for more information on this structure.

For a positive response (**req->resp_err = CNF_RESP_OK**), the request control structure member **resp_info_ptr** is a pointer to the operation specific information structure (**STATUS_RESP_INFO**) for the Status function.

For negative response, or other errors, see **Volume 3 — Module 11 — MMS-EASE Error Handling** for more information.

Return Value: `ST_VOID` (ignored)

Operation Specific Data Structures Used: **STATUS_RESP_INFO**

See page 2-15 for a detailed description of this structure.

4. UnsolicitedStatus Service

This unconfirmed service is used by a node to spontaneously report its status, in an unsolicited manner.

Primitive Level UnsolicitedStatus Operations

The following section contains inform ation on how to use the paired primitive interface for the Unsolicited-Status service. It covers available data structures used by the PPI, and the two primitive level functions that together make up the UnsolicitedStatus service. See **Volume 1 — Module 2 — General Sequence of Events — Unconfirmed Services** for a detailed explanation of the sequence of events that occur within the MMS-EASE environment during unconfirmed services.

- The UnsolicitedStatus service consists of the paired primitive functions of **`mp_ustatus`**, and **`u_ustatus_ind`**.

Data Structures

Request/Indication

This operation specific structure described below is used by the server VMD in issuing the UnsolicitedStatus request (**`mp_ustatus`**). It is received by the client when an UnsolicitedStatus indication (**`u_ustatus_ind`**) is received.

```
struct ustatus_req_info
  {
  ST_INT16 logical_stat;
  ST_INT16 physical_stat;
  ST_BOOLEAN local_detail_pres;
  ST_INT local_detail_len;
  ST_UCHAR local_detail[MAX_STAT_DTL_LEN];
  };
typedef struct ustatus_req_info  USTATUS_REQ_INFO;
```

Fields:

`logical_stat` This required field indicates the logical status of the VMD:

- **`0`** State changes are allowed. All supported services can be used.
- **`1`** No state changes are allowed to services that modify the state of a VMD object.
- **`2`** Limited services are permitted (Status, Identify, and Context Management)

`physical_stat` This required field indicates the physical status of the VMD:

- **`0`** Fully operational.
- **`1`** Partially operational.
- **`2`** Inoperable.
- **`3`** Needs Commissioning (manual intervention may be needed).

`local_detail_pres` **`SD_FALSE`**. Do Not include **`local_detail`** in PDU.

`SD_TRUE`. Include **`local_detail`** in PDU.

`local_detail_len` This indicates the length, IN BITS, of the **`local_detail`**. This cannot be greater than 128 bits in length. **`MAX_STAT_DTL_LEN`** is set by default to be 16 bits.

`local_detail` This implementation-specific bitstring is defined by the particular VMD containing additional data about the status of the VMD.

NOTE: In an unconfirmed service such as UnsolicitedStatus, there are no response and confirm portions of the sequence.

Paired Primitive Interface Functions

mp_ustatus

Usage: This primitive request function sends an UnsolicitedStatus request PDU. It uses the data from a structure of type **USTATUS_REQ_INFO**, pointed to by **info**. It is used to send status information to a remote node in an unsolicited manner (the remote node did not request the status).

Function Prototype:
```
ST_RET mp_ustatus (ST_INT chan, USTATUS_REQ_INFO *info);
```

Parameters:

chan	This integer contains the channel number over which the UnsolicitedStatus PDU is to be sent.
info	This pointer to an Operation Specific data structure of type **USTATUS_REQ_INFO** contains information specific to the Unsolicited Status PDU to be sent.

Return Value:

ST_RET	**SD_SUCCESS**.	No Error.
	<> 0	Error Code.

Corresponding User Confirmation Function: NONE

Operation-Specific Data Structure Used: **USTATUS_REQ_INFO**

See page 2-21 for a detailed description of this structure.

u_ustatus_ind

Usage: This user function is called when an UnsolicitedStatus indication is received by a MMS-User. The information in the Unsolicited Status request reflects the values of the corresponding attributes of the VMD of the sender.

See **Volume 1 — Module 1 — Function Classes — User Indication Function Class** for additional information regarding the handling of indications.

Function Prototype: `ST_VOID u_ustatus_ind (MMSREQ_IND *ind);`

Parameters:

`ind` This is the pointer to the request control structure, **`MMSREQ_IND`**. See **Volume 1 — Module 2 — Indication and Request Control Data Structures** for more information on this structure. The primary item of interest in this structure is **`ind->resp_info_ptr`**. This is a pointer to the operation specific information structure for an UnsolicitedStatus indication (**`USTATUS_REQ_INFO`**). This pointer will always be valid when **`u_ustatus_ind`** is called.

Return Value: `ST_VOID` (ignored)

Operation-Specific Data Structure Used: **`USTATUS_REQ_INFO`**

See page 2-21 for a detailed description of this structure.

5. GetNameList Service

This service is used to request that a responding node return a list (or part of a list) of object names that exist at the VMD.

Primitive Level GetNameList Operations

The following section contains information on how to use the paired primitive interface for the GetNameList service. It covers available data structures used by the PPI, and the four primitive level functions that together make up the GetNameList service. See **Volume 1 — Module 2 — General Sequence of Events — Confirmed Services** for a detailed explanation of the sequence of events that occur within the MMS-EASE environment during confirmed services.

- The GetNameList service consists of the paired primitive functions of **`mp_namelist`**, **`u_namelist_ind`**, **`mp_namelist_resp`**, and **`u_mp_namelist_conf`**.

Data Structures

Request/Indication

This operation specific structure described below is used by the client in issuing a GetNameList request (**`mp_namelist`**). It is received by the server when a GetNameList indication (**`u_namelist_ind`**) is received.

```
struct namelist_req_info
  {
  ST_BOOLEAN   cs_objclass_pres;
  union
    {
    ST_INT16   mms_class;
    struct
      {
      ST_INT   len;
      ST_UCHAR *cs_class;
      } cs;
    } obj;
  ST_INT16     objscope;
  ST_CHAR       dname[MAX_IDENT_LEN+1];
  ST_BOOLEAN   cont_after_pres;
  ST_CHAR      continue_after[MAX_IDENT_LEN+1];
  SD_END_STRUCT
  };
typedef struct namelist_req_info NAMELIST_REQ_INFO;
```

Fields:

`cs_objclass_pres` **`SD_FALSE`**. This indicates to use the **`mms_class`** member of the union **`obj`**. It means the name list will be for an object specified by the MMS standard (ISO 9506).

`SD_TRUE`. This indicates to use the **`cs`** structure member of the union **`obj`**. It means the name list will be for an object specified by a companion standard.

`mms_class` This contains the class of the named object(s) for which a list is to be obtained. Used when **`cs_objclass_pres = 0`**.

`0`	Named Variable
`1`	Scattered Access
`2`	Named Variable List
`3`	Named Type
`4`	Semaphore
`5`	Event Condition
`6`	Event Action
`7`	Event Enrollment
`8`	Journal
`9`	Domain
`10`	Program Invocation
`11`	Operator Station

`cs.len` This indicates the length of the companion standard defined object class pointed to by **`cs.cs_class`**.

`cs.cs_class` This pointer to the ASN.1 data specifies the companion standard defined object for which the name list is to be generated. This data must conform to the appropriate companion standard governing the particular VMD from which the name list is to be obtained.

`objscope` This indicates the scope of the object(s) for which a list is to be obtained:

`VMD_SPEC`. List only VMD Specific names.

`DOM_SPEC`. List only Domain Specific names.

`AA_SPEC`. List only names specific to this association.

`dname` This pointer to the name of the domain is used if **`objscope = DOM_SPEC`**.

`cont_after_pres` **`SD_FALSE`**. Do Not include **`continue_after`** in PDU. Begin the name list response from the beginning of the list.

`SD_TRUE`. Include **`continue_after`** in PDU. Use this when multiple requests must be made to obtain the entire name list because the entire list of names will not fit into a single response.

`continue_after` This pointer to a variable string specifies the name after which the name list in the response should start.

Response/Confirm

This operation specific structure described below is used by the server in issuing a GetNameList response (**`mp_namelist_resp`**). It is received by the client when a GetNameList confirm (**`u_mp_namelist_conf`**) is received.

```
struct namelist_resp_info
  {
  ST_BOOLEAN more_follows;
  ST_INT num_names;
  SD_END_STRUCT
  };
/*ST_CHAR *name_list[]; */
typedef struct namelist_resp_info NAMELIST_RESP_INFO;
```

Fields:

`more_follows` **SD_FALSE**. There are no more names in the **name_list**.

SD_TRUE. There are more names in the name list than can be sent in this response. The requesting node will have to make more requests to obtain the entire name list.

`num_names` This indicates the number of names in this name list response PDU.

`name_list` This is an array of pointers to the names to be sent in this name list. Each name should be a null-terminated, visible string specifying a MMS identifier. They should consist of only numbers, uppercase letters, lower-case letters, the underscore "_," or the dollar sign "$." They should not exceed the length allowed for MMS Identifiers (**MAX_IDENT_LEN** default = 32).

NOTES:

1. Immediately below this structure (contiguous in memory) is a list of character pointers, one for each name in the name list. Essentially the structure and name pointers are allocated in a single call to `chk_malloc` of size: `(sizeof(NAMELIST_RESP_INFO) + num_names * sizeof(ST_CHAR *))`. The requesting user makes use of the contiguous list of pointers as appropriate. The responding user must pass the **mp_namelist_resp** function a pointer to a contiguous block of memory containing the **NAMELIST_RESP_INFO** structure at the top and the list of character pointers below it.

2. FOR RESPONSE ONLY, when allocating a data structure of type **NAMELIST_RESP_INFO**, enough memory must be allocated to hold the information for the name list pointers if **num_names > 0**. The following C statement can be used:

```
info = (NAMELIST_RESP_INFO *) chk_malloc (sizeof(NAMELIST_RESP_INFO) + (num_names
            * (sizeof(ST_CHAR *))));
```

Paired Primitive Interface Functions

mp_namelist

Usage: This primitive request function sends a GetNameList request PDU using the data from a structure of type **NAMELIST_REQ_INFO**, pointed to by **info**. This allows the local node to obtain a list of defined named objects residing at the remote node.

Function Prototype:

```
MMSREQ_PEND *mp_namelist (ST_INT chan,
                          NAMELIST_REQ_INFO *info);
```

Parameters:

chan	This integer contains the channel number over which the GetNameList PDU is to be sent.
info	This pointer to a data structure of type **NAMELIST_REQ_INFO** contains information specific to the GetNameList PDU to be sent.

Return Value:

MMSREQ_PEND *	This pointer to the request control data structure of type **MMSREQ_PEND** is used to send the GetNameList PDU. In case of an error, the pointer is set to null and **mms_op_err** is written with the error code.

Corresponding User Confirmation Function: **u_mp_namelist_conf**

Operation-Specific Data Structure Used: **NAMELIST_REQ_INFO**

See page 2-25 for a detailed description of this structure.

u_namelist_ind

Usage: This user function is called when a GetNameList indication is received by a server node. The contents and operation of this function are user-defined, and depend on the application requirements. To conform to the MMS protocol requirements, the application must do one of the following some time after the indication is received:

1) call the positive response using either:

   a) the primitive response function (**`mp_namelist_resp`**) after the list of defined named objects has been obtained,

   b) the virtual machine response function (**`mv_namelist_resp`**) to have the virtual machine take the requested action and send the response, or

2) call the appropriate primitive negative (error) response function (**`mp_err_resp`**). See **Volume 3 — Module 11 — MMS-EASE Error Handling** for more information on this function.

See **Volume 1 — Module 1 — Function Classes — Primitive User Confirmation Function Class** for additional information regarding recommended handling of indications.

Function Prototype: `ST_VOID u_namelist_ind  (MMSREQ_IND *ind);`

Parameters:

`ind` This pointer to the request control structure, **`MMSREQ_IND`** is described in **Volume 1 — Module 2 — Indication and Request Control Data Structures**. This pointer must be used to send the response to this indication. The primary item of interest in this structure is **`ind->resp_info_ptr`**. This is a pointer to the operation specific information structure (**`NAMELIST_REQ_INFO`**). This pointer will always be valid when **`u_namelist_ind`** is called.

Return Value: `ST_VOID` (ignored)

Operation-Specific Data Structure Used: **`NAMELIST_REQ_INFO`**

See page 2-25 for a detailed description of this structure.

mp_namelist_resp

Usage: This primitive response function sends a GetNameList positive response PDU. It uses the data from a structure of type **NAMELIST_RESP_INFO**, pointed to by **info**. This should be called after the **u_namelist_ind** function is called (a GetNameList indication is received), and after the specified name list has been successfully obtained.

Function Prototype:

```
ST_RET mp_namelist_resp (MMSREQ_IND *ind,
                         NAMELIST_RESP_INFO *info);
```

Parameters:

ind	This pointer to the indication control structure of type **MMSREQ_IND** is received in the indication function, **u_namelist_ind**.
info	This pointer to an Operation Specific data structure of type **NAMELIST_RESP_INFO** contains information specific to the GetNameList response PDU to be sent.

Return Value: ST_RET **SD_SUCCESS**. No Error.

<> 0 Error Code.

Corresponding User Indication Function: **u_namelist_ind**

Operation-Specific Data Structure Used: **NAMELIST_RESP_INFO**

See page 2-26 for a detailed description of this structure.

u_mp_namelist_conf

Usage: This primitive user confirmation function is called when a valid GetNameList confirmation to a **mp_namelist** is received. **resp_err** contains a value indicating whether an error occurred. **resp_info_ptr** may contain data that the remote node sent back in the confirm, or it may contain error information if an error occurred. If no error has occurred (**req->resp_err = CNF_RESP_OK**), GetNameList information is available to the application using the **req->resp_info_ptr** pointer. See **Volume 1— Module 1 — Function Classes — Primitive User Confirmation Function Class** for more information on the recommended handling of confirms.

Note that the function called for this event may be changed if required; however, **u_mp_namelist_conf** is the standard default function.

Function Prototype:

```
ST_VOID u_mp_namelist_conf (MMSREQ_PEND *req);
```

Parameters:

`req` This pointer of structure type **MMSREQ_PEND** is returned from the original **mp_namelist** request. See **Volume 1 — Module 2 — Indication and Request Control Data Structures** for more information on this structure.

For a positive response (**req->resp_err = CNF_RESP_OK**), the request control structure member **resp_info_ptr** is a pointer to the operation specific data structure (**NAMELIST_RESP_INFO**) for the GetNameList function.

For a negative response, or any other error, see **Volume 3 — Module 11— MMS-EASE Error Handling** for guidelines on handling the error.

Return Value: `ST_VOID` (ignored)

Operation-Specific Data Structure Used: **NAMELIST_RESP_INFO**

See page 2-26 for a detailed description of this structure.

Virtual Machine GetNameList Operations

The following section information on how to use the virtual machine interface for the GetNameList service. It covers the function that can be used for the VMI GetNameList service.

- The GetNameList service consists of the virtual machine function **`mv_namelist_resp`**.

There are no virtual machine data structures for this service.

Virtual Machine Interface Function

mv_namelist_resp

Usage: This virtual machine function allows the user to respond to a GetNameList indication (**`u_namelist_ind`**). This can be done without having to create a data structure of type **`NAMELIST_RESP_INFO`** ahead of time. The function retrieves all the names of the specified type by referencing the **`VMD_CTRL`** structure associated with the channel receiving the GetNameList indication, and sends a response with that name list data in it. MMS-EASE assigns all channels to reference the default **`VMD_CTRL`** structure when **`strt_MMS()`** is called. If more than one **`VMD_CTRL`** structure is being used, BE SURE TO SET **`mms_chan_info[chan].objs.vmd`** TO POINT TO THE PROPER VMD STRUCTURE BEFORE CALLING THIS FUNCTION. See **Volume 1 — Module 2 — Channel Information Structure** and page 2-10 for more information.

Function Prototype: `ST_RET mv_namelist_resp (MMSREQ_IND *ind);`

Parameters:

`ind` This pointer to the indication control data structure of the type **`MMSREQ_IND`** is received for the GetNameList indication. See **Volume 1 — Module 2 — Indication and Request Control Data Structures** for more information on this structure.

Return Value: `ST_RET` **`SD_SUCCESS`**. No Error.

`<> 0` Error Code.

6. Identify Service

This service is used to obtain identifying information such a vendor name, and model number, from a responding node.

Primitive Level Identify Operations

The following section contains information on how to use the paired primitive interface for the Identify service. It covers available data structures used by the PPI, and the four primitive level functions that together make up the Identify service. See **Volume 1 — Module 2 — General Sequence of Events — Confirmed Services** for a detailed explanation of the sequence of events that occur within the MMS-EASE environment during confirmed services.

- The Identify service consists of the paired primitive functions of **`mp_ident`**, **`u_ident_ind`**, **`mp_ident_resp`**, and **`u_mp_ident_conf`**.

Data Structures

Response/Confirm

The operation specific structure described below is used by the server in issuing an Identify response (**`mp_ident_resp`**). It is received by the client when an Identify confirm (**`u_mp_ident_ind`**) is received.

```
#define      VEND_LEN     64
#define      MOD_LEN      16
#define      REV_LEN      16

struct ident_resp_info
  {
  ST_CHAR     vend [MAX_VEND_LEN+1];
  ST_CHAR     model[MAX_MOD_LEN+1];
  ST_CHAR     rev  [MAX_REV_LEN+1];
  ST_INT      num_as;
/*MMS_OBJ_ID as   [num_as];                                  */
  SD_END_STRUCT
  };
typedef struct ident_resp_info IDENT_RESP_INFO;
```

Fields:

`vend` This null-terminated character string identifies the organization (e.g., company name) that developed the VMD for which the identifying information is being provided.

`model` This null-terminated character string contains the manufacturer's model number of the system.

`rev` This null-terminated character string contains the revision level of the system specified by the VMD vendor.

NOTE: The MMS specification allows indefinite length strings for these members, but implementor's agreements specify that only 64, 16, and 16 bytes, as indicated in the `#define` statements above, are considered significant.

`num_as` This indicates the number of abstract syntaxes pointed to by **`as`**.

`as` This array of pointers of structure type **`MMS_OBJ_ID`** contains the abstract syntaxes associated with this VMD. This structure may be followed by the abstract syntax's. See **Volume 3 — Module 10 — ISO Object Identifier Structure** for an explanation of this structure.

NOTE: FOR RESPONSE ONLY, when allocating a data structure of type **IDENT_RESP_INFO**, enough memory must be allocated to hold the information for the abstract syntaxes contained in **as**. The following C statement can be used:

```
info = (IDENT_RESP_INFO *) chk_malloc (sizeof(IDENT_RESP_INFO + (num_as *
            (ST_CHAR *))));
```

Paired Primitive Interface Functions

mp_ident

Usage: This primitive request function sends an Identify request PDU. It is used to obtain identifying information from the remote node such as vendor name, model number, and revision number.

Function Prototype: `MMSREQ_PEND *mp_ident (ST_INT chan);`

Parameters:

`chan` This integer contains the channel number over which the Identify request PDU will be sent.

Return Value:

`MMSREQ_PEND *` This pointer to the request control data structure of type **`MMSREQ_PEND`** is used to send the Identify PDU. See **Volume 1 — Module 2 — Indication and Request Control Data Structures** for more information on this structure. In case of an error, the pointer is set to null and **`mms_op_err`** is written with the error code.

Corresponding User Confirmation Function: **`u_mp_ident_conf`**

Operation Specific Data Structure Used: NONE

u_ident_ind

Usage: This user function is called when an Identify indication is received by a server node. The contents and operation of this function are user-defined, and depend on the application requirements. To conform to the MMS protocol, the application must do one of the following some time after the indication is received:

1) call the positive response using either:

   a) the primitive response function (**`mp_ident_resp`**) after the identifying information about the local MMS node has been successfully obtained,

   b) the virtual machine response function (**`mv_ident_resp`**) to have the virtual machine take the requested action and send the response, or

2) call the appropriate primitive negative (error) response function (**`mp_err_resp`**). See **Volume 3 — Module 11 — MMS-EASE Error Handling** for more information on this function.

See **Volume 1 — Module 1 — Function Classes — User Indication Function Class** for additional information regarding the handling of indications.

Function Prototype:

```
ST_VOID u_ident_ind  (MMSREQ_IND *ind);
```

Parameters:

`ind` This is the pointer to the request control structure, **`MMSREQ_IND`**. See **Volume 1— Module 2 — Indication and Request Control Data Structures**. This pointer must be used to send the response to this indication. This pointer will always be valid when **`u_ident_ind`** is called.

Return Value: `ST_VOID` (ignored)

Operation Specific Data Structure Used: NONE

mp_ident_resp

Usage: This primitive response function sends an Identify positive response PDU. It uses the data from a structure of type **IDENT_RESP_INFO**, pointed to by **info**. This should be called after the **u_ident_ind** is called (an Identify indication was received), and after the identifying information about the local MMS node has been successfully obtained.

Function Prototype:

```
ST_RET mp_ident_resp (MMSREQ_IND *ind,
                      IDENT_RESP_INFO *info);
```

Parameters:

`ind` This pointer to the indication control structure, **MMSREQ_IND** is received in the indication service function **u_ident_ind**.

`info` This pointer to an Operation Specific data structure of type **IDENT_RESP_INFO** contains information specific to the Identify response PDU to be sent.

Return Value: `ST_RET` **SD_SUCCESS**. No Error.

<> 0 Error Code.

Corresponding User Indication Function: **u_ident_ind**

Operation-Specific Data Structure Used: **IDENT_RESP_INFO**

See page 2-33 for a detailed description of this structure.

u_mp_ident_conf

Usage: This primitive user confirmation function is called when an Identify confirm to a primitive request (**mp_ident**) is received. **resp_err** contains a value indicating whether an error occurred. **resp_info_ptr** may contain data that the remote node sent back in the confirm, or it may contain error information if an error occurred. If no error has occurred, (**req->resp_err = CNF_RESP_OK**), Identify information is available to the application using the **req->resp_info_ptr** pointer. See **Volume 1 — Module 1 — Function Classes — Primitive User Confirmation Function Class** for more information on recommended handling of confirms.

Note that the function called for this event may be changed if required; however, **u_mp_ident_conf** is the standard default function.

Function Prototype:

```
ST_VOID u_mp_ident_conf (MMSREQ_PEND *req);
```

Parameters:

`req` This pointer to the **MMSREQ_PEND** structure is returned from the original **mp_ident** request. See **Volume 1 — Module 2 — Indication and Request Control Data Structures** for more information on this structure.

For a positive response (**req->resp_err = CNF_RESP_OK**), the request control structure member **resp_info_ptr** is a pointer to the operation specific data structure (**IDENT_RESP_INFO**).

For a negative response, or other error, see **Volume 3 — Module 11 — MMS-EASE Error Handling** for more information.

Return Value: `ST_VOID` (ignored)

Operation-Specific Data Structure Used: **IDENT_RESP_INFO**

See page 2-33 for a detailed description of this structure.

Virtual Machine Identify Operations

The following section contains information on how to use the virtual machine interface for the Identify service. It covers the response data structure and the response function used by the VMI Identify service.

- The Identify service consists of the virtual machine function **mv_ident_resp**.

Data Structures

Response

The following data structure is used by the MMS-EASE virtual machine to send responses to Identify indications using the **mv_ident_resp** function. This data structure should be filled in with the information to be reported in an Identify response before accepting Identify indications, and responding to them using the virtual machine. See page 2-33 for information regarding the format of **IDENT_RESP_INFO**. This structure is allocated during **strt_MMS** and does not exist before that time.

```
extern IDENT_RESP_INFO *mmse_ident_info;
```

Virtual Machine Interface Function

mv_ident_resp

Usage: This virtual machine function allows responding to an Identify indication (**u_ident_ind**) using the information from the **mmse_ident_info** structure of type **IDENT_RESP_INFO** (see above). If this structure is filled out after calling **strt_MMS**, the virtual machine can respond to identify requests; thereby, not having to deal with the operation specific data structures.

Function Prototype: `ST_RET mv_ident_resp (MMSREQ_IND *ind);`

Parameters:

`ind` This pointer to a indication control data structure of the type **MMSREQ_IND** is received for the Identify indication. See **Volume 1 — Module 2 — Request and Indication Control Data Structures** for information on this structure.

Return Value: `ST_RET` **SD_SUCCESS**. No Error.

`<> 0` Error Code.

7. Rename Service

This service is used to request that the identifier of an object be changed to a specified identifier.

Primitive Level Rename Operations

The following section contains information on how to use the paired primitive interface for the Rename service. It covers available data structures used by the PPI, and the four primitive level functions that together make up the Rename service. See **Volume 1 — Module 2 — General Sequence of Events — Confirmed Services** for a detailed explanation of the sequence of events that occur within the MMS-EASE environment during confirmed services.

- The Rename service consists of the paired primitive functions of **`mp_rename`**, **`u_rename_ind`**, **`mp_rename_resp`**, and **`u_mp_rename_conf`**.

Data Structures

Request/Indication

This operation specific structure described below is used by the client in issuing a Rename request (**`mp_rename`**). It is received by the server when a Rename indication (**`u_rename_ind`**) is received.

```
struct rename_req_info
  {
  ST_BOOLEAN   cs_objclass_pres;
  union
    {
    ST_INT16   mms_class;
    struct
      {
      ST_INT   len;
      ST_UCHAR *cs_class;
      } cs;
    } obj;
  OBJECT_NAME  cur_name;
  ST_CHAR       new_ident [MAX_IDENT_LEN+1];
  SD_END_STRUCT
  };
typedef struct rename_req_info RENAME_REQ_INFO;
```

Fields:

`cs_objclass_pres`

`SD_FALSE`. This indicates to use the **`mms_class`** member of the union **`obj`**. It means that the rename will be for an object specified by the MMS standard (ISO 9506).

`SD_TRUE`. This indicates to use the **`cs`** structure member of the union **`obj`**. It means that the rename will be for an object specified by a companion standard.

`mms_class` This indicates the class of the named object for which a rename is to be done. Used when **`cs_objclass_pres = 0`**.

- **`0`** Named Variable
- **`1`** Scattered Access
- **`2`** Named Variable List
- **`3`** Named Type
- **`4`** Semaphore
- **`5`** Event Condition
- **`6`** Event Action
- **`7`** Event Enrollment
- **`8`** Journal
- **`9`** Domain
- **`10`** Program Invocation
- **`11`** Operator Station

`cs.len` This indicates the length of the companion standard defined object class pointed to by **`cs.cs_class`**.

`cs.cs_class` This pointer to ASN.1 data specifies the companion standard for which a rename is to be accomplished. This data must conform to the appropriate companion standard governing the particular VMD for which the rename is to be done.

`curr_name` This structure of type **`OBJECT_NAME`** contains the current name of the object to be renamed. See **Volume 1 — Module 2 — MMS Object Name Structure** for a detailed description of this structure.

`new_ident` This contains the new name of the object to be renamed.

Paired Primitive Interface Functions

mp_rename

Usage: This primitive request function sends a Rename request PDU. It uses the data from a structure of type **RENAME_REQ_INFO**, pointed to by **info**. This function is used to rename an object at the remote node.

Function Prototype:
```
MMSREQ_PEND *mp_rename (ST_INT chan,                RE-
                                    NAME_REQ_INFO *info);
```

Parameters:

chan	This integer contains the channel number over which the Rename PDU is to be sent.
info	This pointer to an Operation Specific data structure of type **RENAME_REQ_INFO** contains information specific to the Rename PDU to be sent.

Return Value:

MMSREQ_PEND *	This pointer to the request control data structure of type **MMSREQ_PEND** is used to send the Rename PDU. In case of an error, the pointer is set to null and **mms_op_err** is written with the error code.

Corresponding User Confirmation Function: **u_mp_rename_conf**

Operation-Specific Data Structure Used: **RENAME_REQ_INFO**

See page 2-41 for a detailed description of this structure.

u_rename_ind

Usage: This user function is called when a Rename indication is received by a server node. The contents and operation of this function are user-defined, and depend on the application requirements. To conform to the MMS protocol requirement, the application must do one of the following some time after the indication is received:

1) call the primitive response function (**`mp_rename_resp`**) after the specified object has been successfully renamed, or

2) call the appropriate primitive negative (error) response function (**`mp_err_resp`**). See **Volume 3 — Module 11 — MMS-EASE Error Handling** for more information on this function.

See **Volume 1 — Module 1 — Function Classes — User Indication Function Class** for additional information regarding the handling of indications.

Function Prototype: `ST_VOID u_rename_ind (MMSREQ_IND *ind);`

Parameters:

`ind` This pointer to the **`MMSREQ_IND`** structure is described in **Volume 1 — Module 2 — Indication and Response Control Data Structures**. This pointer must be used to send the response to this indication. The primary item of interest in this structure is **`ind->resp_info_ptr`**. This is a pointer to the operation specific data structure for a Rename indication (**`RENAME_REQ_INFO`**). This pointer will always be valid when **`u_rename_ind`** is called.

Return Value: `ST_VOID` (ignored)

Operation-Specific Data Structures Used: **`RENAME_REQ_INFO`**

See page 2-41 for a detailed description of this structure.

mp_rename_resp

Usage: This primitive response function sends a Rename positive response PDU. It should be called after the **u_rename_ind** function is called (a Rename indication is received), and after the object has been successfully renamed.

Function Prototype: `ST_RET mp_rename_resp (MMSREQ_IND *ind);`

Parameters:

`ind` This pointer to the indication control structure of type **MMSREQ_IND** is received in the indication service function, **u_rename_ind**.

Return Value: `ST_RET` **SD_SUCCESS**. No Error.

`<> 0` Error Code.

Corresponding User Indication Function: **u_rename_ind**

Operation-Specific Data Structure Used: NONE

u_mp_rename_conf

Usage: This primitive user confirmation function is called when a confirm to a **mp_rename** is received. **resp_err** contains a value indicating whether an error occurred. **resp_info_ptr** may contain data that the remote node sent back in the confirm, or it may contain error information if an error occurred. If no error has occurred (**req->resp_err = CNF_RESP_OK**), rename information is available to the application using the **req->resp_info_ptr** pointer. See **Volume 1 — Module 1 — Function Classes — Primitive User Confirmation Functions Class** for more information on the recommended handling of confirms.

Note that the function called for this event may be changed if desired; however, **u_mp_rename_conf** is the standard default function.

Function Prototype:

```
ST_VOID u_mp_rename_conf (MMSREQ_PEND *req);
```

Parameters:

`req` This pointer to the **MMSREQ_PEND** structure is returned from the original **mp_rename** request function. See **Volume 1 — Module 2 — Indication and Response Control Data Structures** for more information on this structure.

For negative response, or other errors, see **Volume 3 — Module 11 — MMS-EASE Error Handling** for more information.

Return Value: `ST_VOID` (ignored)

Operation Specific Data Structures Used: NONE

8. GetCapabilityList Service

This service is used to request that a responding node return a list (or part of a list) of capabilities defined at the VMD.

Primitive Level GetCapabilityList Operations

The following section contains information on how to use the paired primitive interface for the GetCapability List service. It covers available data structures used by the PPI, and the four primitive level functions that together make up the GetCapabilityList service. See **Volume 1 — Module 2 — General Sequence of Events — Confirmed Services** for a detailed explanation of the sequence of events that occur within the MMS-EASE environment during confirmed services.

- The GetCapabilityList service consists of the paired primitive functions of **`mp_getcl`**, **`u_getcl_ind`**, **`mp_getcl_resp`**, and **`u_mp_getcl_conf`**.

Data Structures

Request/Indication

This operation specific structure described below is used by the client in issuing a GetCapabilityList request (**`mp_getcl`**). It is received by the server when a GetCapabilityList indication (**`u_getcl_ind`**) is received.

```
struct getcl_req_info
  {
  ST_BOOLEAN cont_after_pres;
  ST_CHAR     *continue_after;
  };
typedef struct getcl_req_info GETCL_REQ_INFO;
```

Fields:

`cont_after_pres`	**`SD_FALSE`**. Do not include the **`continue_after`** field in the PDU.
	`SD_TRUE`. Include the **`continue_after`** field in the PDU.
`continue_after`	This pointer to a visible string specifies the capability after which the capability list in the response should start.

Response/Confirm

This operation specific structure described below is used by the server in issuing a GetCapabilityList response (**`mp_getcl_resp`**). It is received by the client when a GetCapabilityList confirm (**`u_mp_getcl_conf`**) is received.

```
struct getcl_resp_info
  {
  ST_BOOLEAN more_follows;
  ST_INT      num_of_capab;
/*ST_CHAR     *capab_list [num_of_capab];                          */
  SD_END_STRUCT
  };
typedef struct getcl_resp_info GETCL_RESP_INFO;
```

Fields:

`more_follows` **SD_FALSE**. No more data follows in this list.

SD_TRUE. More data follows this list. It is used to signify that there are more capabilities than could be sent in this response.

`num_of_capab` This indicates the number of pointers in the **capab_list** array.

`capab_list` This array of pointers points to null-terminated character strings containing the list of capabilities of the VMD being included in this response.

NOTE: FOR RESPONSE ONLY, when allocating a data structure of type **GETCL_RESP_INFO**, enough memory is allocated to hold the information for the list of pointers to the capabilities contained in **capab_list**. The following C language statement can be used:

```
info = (GETCL_RESP_INFO *) chk_malloc(sizeof(GETCL_RESP_INFO) + num_of_capab *
              sizeof(ST_CHAR *));
```

Paired Primitive Interface Functions

mp_getcl

Usage: This primitive request function sends a GetCapabilityList request PDU to a remote node. It uses the data found in a structure of type **GETCL_REQ_INFO** pointed to by **info**. This service is used to obtain the list of capabilities of the VMD at the remote node.

Function Prototype:

```
MMSREQ_PEND *mp_getcl (ST_INT chan,
                       GETCL_REQ_INFO *info);
```

Parameters:

`chan` This is the channel number over which the PDU is to be sent.

`info` This pointer to an Operation Specific data structure of type **GETCL_REQ_INFO** contains information specific to the request PDU to be sent.

Return Value:

`MMSREQ_PEND *` This function returns a pointer to the request control data structure of type **MMSREQ_PEND** used to send the PDU.

Corresponding User Confirmation Function: **u_mp_getcl_conf**

Operation-Specific Data Structure Used: **GETCL_REQ_INFO**

See page 2-47 for a detailed description of this structure.

u_getcl_ind

Usage: This user function is called when a GetCapabilityList indication is received by a server node. The contents and operation of this function are user-defined, and depend on the application requirements. To conform to the MMS protocol requirements, the application must do one of the following some time after the indication is received:

1) call the primitive positive response function (**mp_getcl_resp**) after the specified list of capabilities of the VMD has been successfully obtained, or

2) call the appropriate primitive negative (error) response function (**mp_err_resp**). See **Volume 3 — Module 11 — MMS-EASE Error Handling** for more information on this function.

See **Volume 1 — Module 1 — Function Classes — User Indication Function Class** for additional information on recommended handling of indications.

Function Prototype: `ST_VOID u_getcl_ind (MMSREQ_IND *ind);`

Parameters:

`ind` This is the pointer to the request control structure, **MMSREQ_IND**. See **Volume 1 — Module 2 — Indication and Request Control Data Structures** for more information. This pointer must be used to send the response to this indication. The primary item of interest in this structure is **ind->resp_info_ptr**. This is a pointer to the operation specific structure for this indication (**GETCL_RESP_INFO**). This pointer will always be valid when **u_getcl_ind** is called.

Return Value: `ST_VOID` (ignored)

Operation-Specific Data Structure Used: **GETCL_REQ_INFO**

See page 2-47 for a detailed description of this structure.

mp_getcl_resp

Usage: This primitive response function sends a GetCapabilityList positive response PDU. It uses the data from a structure of type **GETCL_RESP_INFO**, pointed to by **info**. This function should be called as a response to the **u_getcl_ind** function being called (a GetCapabilityList indication is received), and after the specified list of capabilities of the VMD has been successfully obtained.

Function Prototype:

```
ST_RET mp_getcl_resp (MMSREQ_IND *ind,
                      GETCL_RESP_INFO *info);
```

Parameters:

`ind` This pointer to the indication control structure of type **MMSREQ_IND** is received in the indication service function **u_getcl_ind**.

`info` This pointer to an Operation Specific data structure of type **GETCL_RESP_INFO** contains information specific to the response PDU to be sent.

Return Value: `ST_RET` **SD_SUCCESS**. No Error.

`<> 0` Error Code.

Corresponding User Indication Function: **u_getcl_ind**

Operation-Specific Data Structure Used: **GETCL_RESP_INFO**

See page 2-47 for a detailed description of this structure.

u_mp_getcl_conf

Usage: This primitive user confirmation function is called when a confirm to a **mp_getcl** is received. **resp_err** contains a value indicating whether an error occurred. **resp_info_ptr** may contain data that the remote node sent back in the confirm, or it may contain error information if an error occurred. If no error has occurred (**req->resp_err = CNF_RESP_OK**), GetCapabilityList information is available to the application using the **req->resp_info_ptr** pointer. See **Volume 1 — Module 1 — Function Classes — Virtual User Confirm Function Class** for more information on the recommended handling of confirms.

Note that the function called for this event may be changed if required; however, **u_mp_getcl_conf** is the standard default function.

Function Prototype:

```
ST_VOID u_mp_getcl_conf (MMSREQ_PEND *req);
```

Parameters:

`req` This pointer to the **MMSREQ_PEND** structure is returned from the original **mp_getcl** request function. See **Volume 1 — Module 2 — Indication and Response Control Data Structures** for more information on this structure.

For a positive response (**req->resp_err = CNF_RESP_OK**), the request control structure member **resp_info_ptr** is a pointer to the operation specific data structure (**GETCL_RESP_INFO**).

For a negative response, or other errors, see **Volume 3 — Module 11 — MMS-EASE Error Handling** for more information.

Return Value: `ST_VOID` (ignored)

Operation-Specific Data Structure Used: **GETCL_RESP_INFO**

See page 2-47 for a detailed description of this structure.

1. Variable Access And Management Introduction

This portion of MMS-EASE provides services that allow a client to access typed variables defined at the VMD and for a server to respond to the client requests. These are provided by extending the VMD model to include objects and services that operate on these objects. The MMS variable is an abstract element of the VMD capable of providing (when read) or accepting (when written) the value of a real data type.

MMS Variable Objects

Variables at a server application can be read or written by remote client applications. Some of the variables at the server may be known to other applications and available for access, whereas others are for internal use only. Those which are made known to other applications are called **network variables** or **MMS variables**. The means of making network variables known are twofold: either give the variables network names (which may be the same as local program names) and make them public, or make the addresses and types of network variables public. In the first case, the network variables are **named variable** objects; in the second, they are **unnamed**. MMS-EASE supports both. In addition, MMS-EASE also supports other variable-related objects, such as **named variable lists**, and **named types**. As with variables, variable lists and types may be exposed and made public to other applications (in which case they may be named), or they may be for internal use only.

The variable objects supported by MMS-EASE are:

1. **Unnamed Variable Object**. An unnamed variable object describes the access to the real variable by using a device-specific **ADDRESS**. MMS includes unnamed variable objects primarily for compatibility for older devices that are not capable of supporting names. An unnamed variable object is a direct mapping to the underlying real variable that is located at the specified address. The attributes of the unnamed variable object are:

   Address This is the key attribute of the unnamed variable object.

   MMS Deletable This attribute is always FALSE for an unnamed variable object. Unnamed variable objects cannot be deleted because they are a direct representation of the “real” variable located at a specific address in the VMD.

   Type Description This attribute describes the type (format and range of values) of the variable.

2. **Named Variable Object**. The named variable object describes access to the real variable by using a MMS object name. MMS clients only need to know the name of the object in order to access it. Remember that the name of the MMS variable also specifies the scope of the object. In addition to the name, a named variable object has the following attributes:

   MMS Deletable This attribute indicates if access to the variable can be deleted using the DeleteVariableAccess service.

   Type Description This attribute describes the type (format and range of values) of the variable.

   Access Method If the access method is PUBLIC, it means that the underlying address of the named variable object is visible to the MMS client. In this case, the same variable can be accessed as a unnamed variable object.

3. **Named Variable List Object**. The named variable list object provides an access mechanism for grouping both named and unnamed variable objects into a single object for easier access. a named variable list is accessed by a MMS client by specifying the name (which also specifies its scope) of the named variable list. When the VMD receives the Read service request from a client, it reads all the individual objects in the list and returns their value within the individual elements of the named variable list. Because the named variable list object contains independent subordinate objects, a positive confirmation to a Read request for a named variable list may indicate only partial success. The success/failure status of each individual element in the confirmation must be examined by the client to ensure that all the underlying variable objects were accessed without error. In addition to the name and the list of underlying named and unnamed variable objects, named variable list objects also has the following attribute:

   MMS Deletable This attribute indicates if the named variable list can be deleted using the DeleteNamedVariableList service.

4. **Named Type Object**. The named type object provides a mechanism to assign a name to a MMS **TYPE** description. The type of a variable indicates its format and possible range of values that the variable can take. The named type object allows the types of variables to be defined and managed separately. This can be particularly useful for systems that also use the DefineNamedVariable service to define names and types for unnamed variable objects. Other attributes of the named type object include:

   MMS Deletable This attribute indicates if the named type can be deleted using the DeleteNamedType service.

   Type Description This describes the type in terms of complexity (simple, array, or structured) format (integer, floating point, etc.) and the size or range of values (16-bit, 8 characters, etc.).

5. **Scattered Access Object**. The scattered access object is a named list of scattered (non-contiguous) variables. This is similar to **named variable lists**, but treated as single variables of a structured type.

NOTE: Although the MMS-EASE PPI (Paired Primitive Interface) supports the use of all these objects, the VMI (Virtual Machine Interface) concentrates on making **variables**, **types**, and **variable lists** easier to use.

MMS Services For Variable Access

The MMS objects described on the previous page are manipulated by a number of MMS services. MMS-EASE supports all the Variable Access services listed below:

Read This service is used by MMS clients to obtain the values of named, unnamed, or named variable list objects. See page 2-153 for more information.

Write This service is used by MMS clients to change the values of named, unnamed, or named variable list objects. See page 2-173 for more information.

InformationReport This service is used by a VMD to report the values of named, unnamed, or named variable list objects in an unsolicited manner. It is roughly equivalent to sending a Read response to a MMS User without the MMS User having issued the Read request. The InformationReport service can be used to eliminate polling by client applications or as an alarm notification. The VMD could directly report changes in the value of variables, alarms conditions, or even changes in state of the VMD or program invocations to clients by using this service. MMS does not require the use of the InformationReport service for this purpose; it is simply one of the many potential applications of this service. See page 2-193 for more information.

GetVariableAccessAttributes This service is used by MMS clients to obtain the access attributes of a single named or unnamed variable object. The access attributes are the type (either a named type or type description), the MMS deletable attribute and the address for unnamed variables. The address of named variables is optional and is only returned if the address is known and visible. See page 2-205 for more information.

DefineNamedVariable This service allows MMS clients to create a named variable object corresponding to a real VMD variable that can be represented by an unnamed variable object. Once defined, subsequent access to the unnamed variable object can be made using the named variable. This service allows the MMS client to optionally specify a type definition for the named object that may be different from the type inherently defined for the unnamed variable object. See page 2-213 for more information.

DeleteVariableAccess This service is used to delete named variable objects where the MMS deletable attribute is TRUE. This service provides options for deleting only specific named variable objects or all named variable objects of a specified scope (VMD, domain, or AA-specfic). See page 2-219 for more information.

DefineNamedVariableList This service creates a named variable list object. See page 2-227 for more information.

GetNamedVariableListAttributes This service obtains the attributes (i.e., the list of underlying named and unnamed variable objects) of a named variable list object. See page 2-233 for more information.

DeleteNamedVariableList This service deletes a named variable list object. See page 2-241 for more information.

DefineNamedType This service is used by a MMS client to create a new named type by specifying the type name (including scope), MMS deletable, and type description attributes. See page 2-249 for more information.

GetNamedTypeAttributes This service is used by a MMS client to determine the MMS deletable and type description attributes of a specific named type object. See page 2-255 for more information.

DeleteNamedType This service is used to delete an existing named type where the MMS deletable attribute is TRUE. The service provides options for deleting only specific named type objects or all named type objects of a specified scope (VMD, domain, or AA-specific). See page 2-263 for more information.

Client And Server Functionality

Often, the application will act as a server sometimes, and a client other times. Its own network variables will be available to be read and written by other applications. However, it will sometimes request to read or write other application's network variables. MMS-EASE allows an application to act as a server, client, or both, concerning the variable access services. There are functions available at two levels (the PPI and VMI). They allow an application to request to read and write other application's variables and receive the confirmations to those requests. They also allow an application to respond to other applications' read and write indications. MMS-EASE supports this functionality for other related services, such as defining variable objects, and getting information on them, but Read and Write are the main operations.

The VMI functions are much easier to use, but they require configuration ahead of time using VMI support functions. Some VMI support functions can be used with the PPI to simplify that interface, but most are for use only with the Virtual Machine Interface. In addition, there are several data structures that are used for inputting information to PPI and VMI functions, and for storing configuration information for the VMI. Explaining these goes hand in hand with explaining the MMS-EASE Variable Access functions, and both are presented in this Module.

Data Structures For Variable Access

There are several data structures in which configuration information is stored for use by VMI functions. These are **NAMED_VAR** for variable data, **NAMED_VAR_LIST** for variable list data, and **NAMED_TYPE** for type data. These structures are used for storing information about local named variables and types only. VMI functions are provided for building these structures and managing them (the Virtual Machine Database Management functions), and the Virtual Machine Response functions use them when the application is acting as a server and responding to other applications' requests. The **MV_VARDESC** structure contains information about remote named variables and is used by the Virtual Machine Request functions. Both **MV_VARDESC** and **NAMED_VAR** structures reference **NAMED_TYPE** structures, since all variables, both local and remote, have types associated with them.

Besides the VMI data structures, there are several data structures used for inputting information to and receiving information from the PPI functions. These are:

- the address structures **UNCONST_ADDR** and **VAR_ACC_ADDR**;
- the variable specification structures **VARIABLE_DESC** and **VARIABLE_SPEC**;
- the type specification structures **VAR_ACC_TSPEC**, **RUNTIME_TYPE**, and **SCATTERED_ACCESS**;
- the variable list structures **VARIABLE_LIST**, **VAR_ACC_SPEC**;
- the alternate access data structures **ALT_ACCESS**, **ALT_ACC_EL**, **ALTERNATE_ACCESS**, and **RT_AA_CTRL**;
- the data access structures **ACCESS_RESULT**, **WRITE_RESULT**, **SCATTERED_ACCESS**, and **VAR_ACC_DATA**.

These structures are all interrelated, and are explained in detail later in this section. Note that the address structures are also used by the VMI structure **NAMED_VAR**, since local variables have explicit addresses associated with them.

Variables And Types

Every variable, by nature, has a type associated with it. The data residing in a variable must be interpreted according to the variable's type. The MMS-EASE Virtual Machine Interface must always know the variable type (that is, it must be able to get at the appropriate **NAMED_TYPE** structure) in order to do anything with the variable's data. What the virtual machine does is convert data from the local representation of a type to the network representation (the format it is in while being transmitted over the network), and visa versa. Without knowing the type (and local address) of the data, it could not do this conversion. MMS requires all data to be represented in a standard format during transmission (according to the ASN.1 Encoding Rules of ISO 8825), which helps make OSI networks truly open networks.

Specifying types is not a trivial matter, especially with MMS. So MMS-EASE provides a simple ASCII-based language for specifying types. The user application can specify types intuitively with this language, and a Virtual Machine Support function puts this type information into the ASN.1-encoded form needed by MMS. It is then transformed into an internal runtime format. This internal format is used by MMS-EASE in constructing ASN.1 encodings from local representation. It also decodes ASN.1 data to local representation, and is used for extensive checking for data type validity. MMS-EASE maintains a local database for the network type definitions, and these types are referenced by type name. Components of this database are accessible for customizing data handling, if required. MMS-EASE also provides a set of predefined simple types to use.

For local named variables, the Virtual Machine Response functions first locate the appropriate **NAMED_VAR** structure(s). Since each of functions points to a **NAMED_TYPE** structure, they can get the required type information. The Virtual Machine Request functions either require a pointer to a **NAMED_TYPE** structure to be input directly, or it is input directly through the **MV_VARDESC** structure, which names the local type and allows the Virtual Machine to locate the **NAMED_TYPE** structure. For Virtual Machine Request functions, remote variables are being read, but their types must be known (as locally-defined types) in order that the Virtual Machine can convert outgoing or incoming data correctly. The Virtual Machine Data Conversion representation and the ASN.1-encoded format. These functions are used internally by other VMI functions, and applications that use the VMI exclusively do not need to call them directly. They are exposed to make the PPI easier to use, when an application must use the PPI functions directly.

Alternate Access

Alternate Access provides an alternate view of a variable's type. It is used to restrict access to a subset of a range of possible values of the variable or in other words provide partial access to a variable.

2. Virtual Machine Interface

The Virtual Machine Interface can take care of access for variables located in the host on which MMS-EASE is running. In other words, it takes care of local management only. VMI can be used to automatically read and write variables, define types of variables, delete types simply by calling the appropriate virtual request or response function. In addition, several support functions are provided for handling variable definitions, variable list definitions, and type definitions for strictly local variables. It provides the following functionality:

- Database maintenance for variable names, variable list names, and type names.
- Automatic handling of most variable access services.
- Support for named variables, lists of named variables, named variable lists
- Arbitrarily Complex types supported
- Comprehensive support for Alternate Access
- Variable names defined locally or remotely.
- Types defined locally or remotely.
- VMI handles local ⇔ ASN.1 ⇔ runtime translation of all data types
- User-defined variable, variable list, and type names are supported.

NOTES:

1. All data handling by VMI requires defined named types
2. No scattered access is supported by the VMI.
3. The virtual machine allows specifying the physical address of variables before being read or written. MMS-EASE calls the address resolution function, **`u_get_named_addr`**, when it needs to know the physical address of a named variable.

Virtual Machine Database

The following section includes explanations of the data structures and global variables that make up the Virtual Machine Database.

Named Variable Control Structure

This structure is used to keep track of named variables. Local network variables are maintained in a local database containing type, address, and general variable handling information. To create a new named variable, the support function **`ms_add_named_var`** is used. This function then allocates a structure of this type, and provides a pointer to it. Some VMI functions make use of this database to simplify responding.

```
struct named_var
  {
  DBL_LNK      link;
  ST_CHAR      varname[MAX_IDENT_LEN+1];
  ST_BOOLEAN   invalid;
  NAMED_TYPE   *type;
  ST_BOOLEAN   deletable;
  ST_UCHAR     rd_pro;
  ST_UCHAR     wr_pro;
  VAR_ACC_ADDR addr;
  ST_CHAR       *(*read_ind_fun)(ST_CHAR *src, ST_INT len);
  ST_INT       (*write_ind_fun)(ST_CHAR *src, ST_CHAR *dest, ST_INT len);
  SD_END_STRUCT
  };
typedef struct named_var NAMED_VAR;
```

Fields:

`link` *** FOR INTERNAL USE ONLY, DO NOT USE ***

`varname` This contains the name of the variable. This must be Visible String of no longer than 32 characters existing only of numbers (0-9), upper case letters (A-Z), lower case letters (a-z), and the **_** and **$** characters. It must not start with a number or contain spaces.

`invalid` **SD_FALSE**. Variable is invalid. A variable can become invalid if the underlying address, or type definition becomes invalid, or is deleted.

SD_TRUE. Variable is not invalid.

NOTE: MMS-EASE does not use this member of the **NAMED_VAR** structure. It is intended for user application use.

`type` This pointer to a structure of type **NAMED_TYPE** contains the type specification for this variable. See page 2-54 for more information on this structure.

`deletable` **SD_TRUE**. This variable can be deleted over the network.

SD_FALSE. This variable cannot be deleted over the network.

NOTE: MMS-EASE will delete a named variable using **ms_del_named_var**. Local deletions are allowed. However, attempting to delete a named variable using **mv_delvar_resp** will result in failure if **deletable = SD_FALSE**. Network deletions are not allowed when **deletable = SD_FALSE**.

`rd_pro` This is compared to the privilege allowed for a given remote communications entity so that read access to this variable can be controlled. This variable is not currently used by MMS-EASE.

`wr_pro` This is compared to the privilege allowed for a given remote communications entity so that write access to this variable can be controlled. This variable is not currently used by MMS-EASE.

`addr` This structure of type **VAR_ACC_ADDR** contains the address specification for the variable. The address of the variable can be stored here it you want to have that address known to other nodes on the network. Please note that the actual physical address of the variable is determined by the address resolution function **u_get_named_addr**. The address, visible from the network placed in this variable, need not be the actual physical address of the variable. See page 2-150 for more information on the **VAR_ACC_ADDR** structure.

(*read_ind_fun) (*write_ind_fun)

These two elements of the **NAMED_VAR** structure can be written by the user to point to address resolution functions that perform the actual read or write when necessary. The VMI functions for processing Read and Writes Indications also invoke these function pointers. This allows creating separate functions that perform the actual reads or writes based upon the specific variable being read or written. When naming a variable, MMS-EASE writes these function pointers with the values contained in the **NAMED_TYPE** structure corresponding to the variable type being named. NORMALLY THESE FUNCTION POINTERS DO NOT NEED TO BE CHANGED IF YOU ARE USING A TYPE DEFINITION THAT COMES PREPROGRAMMED. However, if you want a specific variable to be treated differently than other variables of the same type, a different function pointer may be written into the **NAMED_VAR** structure for that variable after it has been defined. This can be useful for triggering events based upon when certain variables are accessed. For example, should the Named-Variable represent memory that resides in another process, in a protected kernel, or must be accessed through a serial or other network interface, these functions allow the application to supply the code that performs that type of access. For variables that reside in the data space of the application, the default functions perform the service of Read and Write. They are shown on the next page.

```
/*********************************************************************/
/*                  rdind_fun                                        */
/* simple read_ind fun, no action necessary                          */
/*********************************************************************/

ST_CHAR *rdind_fun (ST_CHAR *addr, ST_INT len)
  {
/* assuming that the addr argument references                        */
/* the beginning location of the variable                            */
  return (addr);
  }

/*********************************************************************/
/*                  wrind_fun                                        */
/* simple write indication function, make simple data block move     */
/*********************************************************************/

ST_RET wrind_fun (ST_CHAR *data, ST_CHAR *addr,     ST_INT len)
  {
  memcpy (addr,data,len);
  return (SD_SUCCESS);
  }
```

See the description of the address resolution functions starting on page 2-80 for more information.

Named Variable List Control Structure

The structure shown on the next page is used to keep track of named variable lists. To create a new named variable list, the support function **ms_add_nvlist** is used. This function then allocates a structure of this type, and provides a pointer to it. Some VMI functions make use of this database to simplify responding.

```
struct named_var_list
  {
  DBL_LNK       link;
  ST_CHAR       name[MAX_IDENT_LEN+1];
  ST_BOOLEAN    deletable;
  ST_INT        num_vars;
/*VARIABLE_LIST var_list [num_of_variables];              */
  SD_END_STRUCT
  };
typedef struct named_var_list NAMED_VAR_LIST;
```

Fields:

`link` *** FOR INTERNAL USE ONLY, DO NOT USE ***

`name` This contains the variable list name. This must be Visible String of no longer than 32 characters existing only of numbers (0-9), upper case letters (A-Z), lower case letters (a-z), and the **_** and **$** characters. It must not start with a number or contain spaces.

`deletable` **SD_TRUE**. This named variable list can be deleted over the network.

SD_FALSE. This named variable list cannot be deleted over the network.

NOTE: MMS-EASE will delete a named variable list using **ms_del_nvlist**. In other words, local deletions are allowed. However, attempting to delete a named variable list using **mv_delvlist_resp** will result in failure if **deletable = SD_FALSE**. Network deletions are not allowed when **deletable = SD_FALSE**.

`num_vars` This is the number of variables contained in the NamedVariableList.

`var_list` This structure of type **`VARIABLE_LIST`** contains information on a variable and any alternate access on that variable in the list of variables to be accessed. See page 2-150 for more information on this structure.

Named Type Control Structure

This is the structure used by MMS-EASE to keep track of a named variable type. To add a named type, the **`ms_add_named_type`** support function is used. Some of the elements can be written by your application program to customize the type that was just defined. It is recommended that the appropriate support function be used to manipulate this structure wherever possible. DO NOT ACCESS MEMBERS OF THIS STRUCTURE THAT ARE INDICATED BY **`/* DO NOT USE */`**. In addition, there are other members of this structure (such as **`erased`**, **`nref`**, **`asn1len`**, **`asn1ptr`**, and **`blocked_len`**) that can be accessed, but should not be modified.

```
struct named_type
  {
  DBL_LNK      link;
  ST_CHAR       type_name[MAX_IDENT_LEN+1];
  ST_BOOLEAN   deletable;
  ST_UCHAR     protection;
  ST_BOOLEAN   erased;
  ST_INT       nref;
  RUNTIME_TYPE *rt_head;
  ST_INT       rt_num;
  ST_INT       asn1len;
  ST_UCHAR     *asn1ptr;
  ST_INT       blocked_len;
  ST_CHAR       *(*read_ind_fun)(ST_CHAR *src, ST_INT len);
  ST_INT        (*write_ind_fun)(ST_CHAR *src, ST_CHAR *dest, ST_INT len);
  };
typedef struct named_type NAMED_TYPE;
```

Fields:

`link` *** FOR INTERNAL USE ONLY, DO NOT USE ***

`typename` This contains the name of the type. This must be Visible String of no longer than 32 characters existing only of numbers (0-9), upper case letters (A-Z), lower case letters (a-z), and the **_** and **$** characters. It must not start with a number or contain spaces.

`deletable` **`SD_TRUE`**. This type can be deleted over the network.

`SD_FALSE`. This type cannot be deleted over the network.

NOTE: MMS-EASE will delete a named type using **`ms_del_named_type`**. In other words, local deletions are always allowed. However, the VM function **`mv_deltype_resp`** will not delete it if **`deletable = SD_FALSE`**. In that case, network deletions are not allowed.

`protection` This variable is compared to the privilege allowed for a given remote communications entity so that access to variables of this type can be controlled. MMS-EASE does not use the **`protection`** member of this structure. This member is only here for use by the user's application program.

`erased` If this flag is set equal to **`SD_TRUE`**, then it means that this type definition has been deleted and the Named Type no longer exists from a Network viewpoint. However, this type is still used if a variable referencing it still exists. In this case, the type remains in existence as long as any variable still references it. When there are no variables that reference it any longer (**`nref = 0`**), this type will automatically be deleted.

`nref` This contains a count of the number of defined named variables that use this type definition.

`rt_head` This pointer to a structure of type **`RUNTIME_TYPE`** contains the beginning of the runtime type definition. See page 2-99 for a definition of this structure.

`rt_num` This indicates the number of runtime type blocks (**`rt_blocks`**) in the runtime type definition.

`asn1len` This variable contains the length of the ASN.1 type definition in bytes.

`asn1ptr` This variable contains a pointer to the ASN.1 type definition. Please Refer to **Volume 1 — Appendix I** for more information on the ASN.1 Type Specification, and how to create it.

`blocked_len` This variable contains the length in bytes of the size of the local representation of the variable. When a write occurs, this is the number of bytes transferred from the PDU buffer into the local variable space. This variable is used primarily for the **`u_write_ind`** and **`u_read_ind`** functions.

`(*read_ind_fun)  (*write_ind_fun)`

These two members of the **`NAMED_TYPE`** structure can be used by the application program to point to address resolution functions that perform the actual read or write. This allows creating separate functions to perform the actual reads or writes based upon the type of variable being read or written. The type definitions, that come preprogrammed with MMS-EASE, already have the appropriate (**`*read_ind_fun`**) and (**`*write_ind_fun`**) function pointers written into the **`NAMED_TYPE`** structure corresponding to that type. When a new type is defined, a pointer to a function to handle variables whose storage exists locally (a memory copy function for the (**`*write_ind_fun`**), and a function that simply returns the address for the (**`*read_ind_fun`**)) is written into these members. This means that it is only necessary to change these functions if any special processing is performed, or if the variables do not exist in local memory.

Anytime a new variable name is defined using a call to **`ms_add_named_var`**, MMS-EASE places the pointers to the **`read_ind_fun`** and the **`write_ind_fun`** found in this **`NAMED_TYPE`** structure into the appropriate **`NAMED_VAR`** structure for the variable being defined. See page 2-59 for more information. This is so all access to that variable will use the function pointers written into the **`NAMED_TYPE`** structure for that type. See the description of the address resolution functions on page 2-82 for more information.

Alternate Access Control Structures

These structures are used by MMS-EASE to implement alternate access. In MMS, all variables have a type associated with them. Alternate access allows specifying an alternate view of a variable's type. MMS-EASE uses alternate access to restrict access to a subset of the range of possible variable values. In various variable access services, alternate access is represented by the data structures shown below. **`ALT_ACCESS`** is the data structure that is normally passed to the various MMS-EASE VMI functions. **`ALT_ACC_EL`** is the basic structure for defining a variable using alternate access.

`ALT_ACCESS`

```
struct alt_access
  {
  ST_INT     num_aa;
  ALT_ACC_EL *aa;
  };
typedef struct alt_access ALT_ACCESS;
```

Fields:

`num_aa` This indicates the number of alternate access elements present.

`aa` This pointer of structure type **`ALT_ACC_EL`** contains the alternate access elements.

ALT_ACC_EL

```
struct alt_acc_el
  {
  ST_BOOLEAN  comp_name_pres;
  ST_CHAR     comp_name[MAX_IDENT_LEN+1];
  ST_INT      sel_type;
  union
    {
    ST_CHAR   component[MAX_IDENT_LEN+1];
    ST_UINT32 index;
    struct
      {
      ST_UINT32 low_index;
      ST_UINT32 num_elmnts;
      } ir;
    } u;
  };
typedef struct alt_acc_el ALT_ACC_EL;
```

Fields:

`comp_name_pres` **SD_FALSE**. Do Not include **comp_name** in PDU.

SD_TRUE. Include **comp_name** in PDU.

`comp_name` This specifies the named component of the alternate access for use in deriving a type.

`sel_type` This indicates the type of alternate access selection.

AA_COMP. This selects a single named component of the structure as specified in the **component** parameter.

AA_INDEX. This selects a single array element as specified in the **index** parameter. The element specified is the element in the derived array type to be accessed and is relative to array element 0.

AA_INDEX_RANGE. This selects the range of array of elements as specified in the **low_index** and **num_elmnts** parameters. The low index is relative to array element 0.

AA_ALL. This selects all the array elements of the structure.

AA_COMP_NEST. This selects a single component of the derived type but implies that further AlternateAccess will take place relative to the named component.

AA_INDEX_NEST. This selects a single array element from the derived type but implies that further AlternateAccess will take place relative to the array element.

AA_INDEX_RANGE_NEST. This selects a range of array elements from the derived type but implies that further AlternateAccess will take place relative to each element in the range.

AA_ALL_NEST. This selects all array of elements from the derived type but implies that further AlternateAccess will take place on each of the array elements.

AA_END_NEST. This designates the end of a nesting level. Each **AA_COMP_NEST**, **AA_INDEX_NEST**, **INDEX_RANGE_NEST**, **AA_ALL_NEST**, will have a corresponding **AA_END_NEST**.

`component` This indicates the name of the component when the **sel_type** is set to **AA_COMP** or **AA_COMP_NEST**.

`index` This is the value of the array element when the **sel_type** is set to **AA_INDEX** or **AA_INDEX_NEST**.

`low_index` This is the value of the starting element in the array when the **sel_type** is set to **AA_INDEX_RANGE** or **AA_INDEX_RANGE_NEST**.

`num_elmnts` This indicates the number of elements or range for the AlternateAccess relative to the low_index when the **`sel_type`** of **`AA_INDEX_RANGE`** or **`AA_INDEX_RANGE_NEST`**.

Alternate Access Control Variables

The following variables are used to control functionality in Alternate Access:

`extern ST_BOOLEAN m_alt_acc_packed;`

This variable is a flag that is used to determine whether the local data format is in “packed” or “original” type form. This is used in all conversions to and from local data when the Alternate Access Data library is used. The default value is **`SD_FALSE`**.

`extern ST_INT m_max_dec_aa;`

This variable determines the size of the table allocated for decoding ASN.1 Alternate Access. Limiting the size of the table limits the complexity of the Alternate Access that can be handled. The default value is 50.

`extern ST_INT m_hw_dec_aa;`

This variable indicates the “high water” mark of how many Alternate Access were actually required. If necessary, it can be used as empirically gathered data related to **`m_max_dec_aa`**. **`m_max_dec_aa`** can be modified according to the noted Alternate Access value to tune memory usage.

`extern ST_INT m_max_rt_aa_ctrl;`

When performing ASN.1 ⇔ local data format conversion, the runtime type is “unrolled” and one element is required for each data element and structure or array start and end. This variable determines the size of the table allocated for this operation. Limiting the size of the table limits the size and complexity of data types that are handled. The default is 1000.

`extern ST_INT m_hw_rt_aa_ctrl;`

This variable indicates the “high water” mark of how many elements were actually required for **`m_max_rt_aa_ctrl`**. If necessary, it can be used as empirically gathered data related to **`m_max_rt_aa_ctrl`**. **`m_max_rt_aa_ctrl`** can be modified according to the noted AA value to tune memory usage.

Global Variables

The following variables are used with the Virtual Machine regarding variable access definition:

`extern ST_INT  max_mmsease_vars;`

This variable indicates the maximum number of named variables allowed by the virtual machine to be defined locally. The default value for this number is 500.

`extern ST_INT  mms_var_count;`

This variable keeps a running count of all the named variables currently defined for all defined VMDs.

`extern ST_INT  max_mmsease_nvlists;`

This variable indicates the maximum number of named variable lists allowed by the virtual machine to be defined locally. The default value for this number is 500.

`extern ST_INT  mms_nvlist_count;`

This variable keeps a running count of all the named variable lists currently defined for all defined VMDs.

The following variables are used with the Virtual Machine regarding type definition:

extern ST_INT max_mmsease_types;

This variable indicates the maximum number of named types allowed by the virtual machine to be defined locally. The default value for this number is 500.

extern ST_INT mms_type_count;

This variable keeps a running count of all the named types currently defined for all defined VMDs.

extern ST_BOOLEAN m_calc_rt_size;

This variable is a flag that is used to calculate the number of runtime type blocks (**rt_blocks**). This is used in the **ms_asn1_to_runtime** function.

extern ST_INT m_rt_type_limit;

This variable keeps a running count of the defined limit of runtime types. The default is 50.

Named Variable Manipulation Functions

The functions described next are called by the user's application program and perform various support functions required by MMS-EASE. They are used to access and update the virtual machine database of named variables.

ms_add_var

Usage: This support function allocates a structure of type **NAMED_VAR**. It inserts a named variable into the alphabetized list of named variables contained in the **var_list** member of the appropriate **DOMAIN_OBJS** structure. This structure could reside in a **NAMED_DOM_CTRL** or **VMD_CTRL** structure. This function selects the standard (***write_ind_fun**) and (***read_ind_fun**) function pointers for the new **NAMED_VAR** structure found in the **NAMED_TYPE** structure corresponding to the named type. It requires that a named type exist before calling it, and that a pointer to this named type must be used. It also verifies that a variable of the same name does not already exist in the same domain. After calling this function, the other members of the **NAMED_VAR** structure can be modified if necessary. If more than one **VMD_CTRL** structure is being used locally, be sure to set **m_vmd_select** to point to the appropriate VMD control structure that contains this domain before calling this function.

Function Prototype:

```
NAMED_VAR *ms_add_var (DOMAIN_OBJS *dom,
                       ST_CHAR *name,
                       NAMED_TYPE *type,
                       VAR_ACC_ADDR *addr,
                       ST_INT chan);
```

Parameters:

`dom` This is a pointer to the domain object structure of type **DOMAIN_OBJS** into which the named variable is to be added.

`name` This is a pointer to the name of the variable being created.

`type` This structure of type **NAMED_TYPE** points to an already defined type.

`addr` This structure of type **VAR_ACC_ADDR** points the address of the variable.

`chan` This is the channel number if the named type to be used only if application-association specific. Used only if **type.scope = AA_SPEC(1)**.

Return Value:

`NAMED_VAR *` This returns a pointer to the variable definition structure of type **NAMED_VAR** for the variable just added. This structure is a member of the variable list pointed to by the **var_list** member of the appropriate **DOMAIN_OBJS** structure. In case of an error, the pointer is set to null and **mms_op_err** is written with the error code.

Data Structures Used:

See **Module 4 — VMD Control** starting on page 2-10, and **Module 6 — Domain Management** starting on page 2-277 for more information.

ms_add_named_var

Usage: This support function allocates a structure of type **NAMED_VAR**. It inserts a named variable into the alphabetized list of named variables contained in the **var_list** member of the appropriate **DOMAIN_OBJS** structure. This structure could reside in a **NAMED_DOM_CTRL** or **VMD_CTRL** structure. This function selects the standard (***write_ind_fun**) and (***read_ind_fun**) function pointers for the new **NAMED_VAR** structure found in the **NAMED_TYPE** structure corresponding to the named type. It requires that a named type exist before calling it, and also verifies that a variable of the same name does not already exist in the same domain.

This function differs from the **ms_add_var** function in that a pointer to the **NAMED_TYPE** structure is not needed because a call to **ms_find_named_type_objs** is made internally. It finds the appropriate type using the information in the **OBJECT_NAME** structure and then calls the **ms_add_var** function. This function is used if a pointer to the named type structure is not known.

After calling this function, the other members of the **NAMED_VAR** structure can be modified if necessary. If more than one **VMD_CTRL** structure is being used locally, be sure to set **m_vmd_select** to point to the appropriate VMD control structure that contains this domain before calling this function.

Function Prototype:

```
NAMED_VAR *ms_add_named_var (DOMAIN_OBJS *dom,
                             ST_CHAR *name,
                             OBJECT_NAME *type,
                             VAR_ACC_ADDR *addr,
                             ST_INT chan);
```

Parameters:

dom	This is a pointer to the domain object structure of type **DOMAIN_OBJS** into which the named variable is to be added.
name	This is a pointer to the name of the variable being created.
type	This pointer to a structure of type **OBJECT_NAME** contains the object name of an already defined type.
addr	This structure of type **VAR_ACC_ADDR** points to the address of the variable.
chan	This is the channel number if the named type to be used is Application-Association (AA-) specific. Used only if **type.scope = AA_SPEC (1)**.

Return Value:

NAMED_VAR *	This returns a pointer to the variable definition structure of type **NAMED_VAR** for the variable just added. This structure is a member of the variable list pointed to by the **var_list** member of the appropriate **DOMAIN_OBJS** structure. In case of an error, the pointer is set to null and **mms_op_err** is written with the error code.

Data Structures Used:

See **Module 4 — VMD Control** starting on page 2-10, and **Module 6 — Domain Management** starting on page 2-277 for more information.

ms_add_nvlist

Usage: This support function allocates a structure of type **NAMED_VAR_LIST**. It inserts a named variable list into the alphabetized list of named variable lists contained in the **var_list** member of the appropriate **DOMAIN_OBJS** structure. This structure could reside in a **NAMED_DOM_CTRL** or **VMD_CTRL** structure. After calling this function, the other members of the **NAMED_VAR_LIST** structure can be modified if necessary. If more than one **VMD_CTRL** structure is being used locally, be sure to set **m_vmd_select** to point to the appropriate VMD control structure that contains this domain before calling this function.

Function Prototype:

```
NAMED_VAR_LIST *ms_add_nvlist (DOMAIN_OBJS *dom,
                               ST_CHAR *name,
                               ST_INT num_vars,
                               VARIABLE_LIST *var_list);
```

Parameters:

dom	This is a pointer to the domain object structure of type **DOMAIN_OBJS** into which the named variable list is to be added.
name	This is a pointer to the variable list name being created.
num_vars	This indicates the number of elements in the array pointed to by **var_list**.
var_list	This is a pointer to an array of **VARIABLE_LIST** structures.

Return Value:

NAMED_VAR_LIST *	This returns a pointer to the variable definition structure of type **NAMED_VAR_LIST** for the named variable list just added. This structure is a member of the variable list pointed to by the **var_list** member of the appropriate **DOMAIN_OBJS** structure. In case of an error, the pointer is set to null and **mms_op_err** is written with the error code.

Data Structures Used:

See **Module 4 — VMD Control** starting on page 2-10, and **Module 6 — Domain Management** starting on page 2-277 for more information.

ms_del_all_named_vars

Usage: This support function deletes ALL named variables from a **DOMAIN_OBJS** structure. If more than one **VMD_CTRL** structure is being used locally, be sure to set **m_vmd_select** to point to the appropriate VMD control structure that contains this domain before calling this function.

Function Prototype:

```
ST_VOID ms_del_all_named_vars (DOMAIN_OBJS *dom);
```

Parameters:

dom	This pointer to the domain object structure of type **DOMAIN_OBJS** contains the set of named variables to be deleted.

Return Value: ST_VOID (ignored)

Data Structures Used:

See **Module 4 — VMD Control** starting on page 2-10, and **Module 6 — Domain Management** starting on page 2-277 for more information.

ms_del_all_nvlists

Usage: This support function deletes ALL named variable lists from a **DOMAIN_OBJS** structure. If more than one **VMD_CTRL** structure is being used locally, be sure to set **m_vmd_select** to point to the appropriate VMD control structure that contains this domain before calling this function.

Function Prototype:

```
ST_VOID ms_del_all_nvlists (DOMAIN_OBJS *dom);
```

Parameters:

`dom` This pointer to the domain object structure of type **DOMAIN_OBJS** contains the set of named variable lists to be deleted.

Return Value: `ST_VOID` (ignored)

Data Structures Used:

See **Module 4 — VMD Control** starting on page 2-10, and **Module 6 — Domain Management** starting on page 2-277 for more information.

ms_del_named_var

Usage: This function deletes a named variable from a **DOMAIN_OBJS** structure. If more than one **VMD_CTRL** structure is being used locally, be sure to set **m_vmd_select** to point to the appropriate VMD control structure that contains this domain before calling this function.

Function Prototype:

```
ST_RET ms_del_named_var (DOMAIN_OBJS *dom,
                         ST_CHAR *name);
```

Parameters:

`dom` This is a pointer to the domain object structure of type **DOMAIN_OBJS** from which the named variable is to be deleted.

`name` This is a pointer to the name of the variable being deleted.

Return Value: `ST_RET` **SD_SUCCESS**. No Error.

`<> 0` Error Code.

Data Structures Used:

See **Module 4 — VMD Control** starting on page 2-10, and **Module 6 — Domain Management** starting on page 2-277 for more information.

ms_del_named_var_obj

Usage: This function deletes a named variable defined by the **OBJECT_NAME** element **obj** and returns a pointer to it. It differs from the **ms_del_named_var** function in that a pointer to the **DOMAIN_OBJS** structure is not needed because a call to **ms_find_domain_objs** is made internally. It finds the appropriate domain using the information in the **OBJECT_NAME** structure and then calls the **ms_del_named_var** function. This function is used if a pointer to the domain object structure is not known.

Function Prototype:

```
ST_RET ms_del_named_var_obj (OBJECT_NAME *obj,
                             ST_INT chan);
```

Parameters:

`obj` This is a pointer to the object name structure of type **DOMAIN_OBJS** from which the named variable is to be deleted.

`chan` This indicates the channel number and is only used if the Named Variable defined by **obj** is Application-Association (AA-) specific.

Return Value: `ST_RET`

SD_SUCCESS. No Error.

`<> 0` Error Code.

Data Structures Used:

See **Module 6 — Domain Management** starting on page 2-277 for more information on the **DOMAIN_OBJ** structure.

ms_del_nvlist

Usage: This function deletes a named variable list from a **DOMAIN_OBJS** structure. If more than one **VMD_CTRL** structure is being used locally, be sure to set **m_vmd_select** to point to the appropriate VMD control structure that contains this domain before calling this function.

Function Prototype:

```
ST_RET ms_del_nvlist (DOMAIN_OBJS *dom, ST_CHAR *name);
```

Parameters:

dom	This is a pointer to the domain object structure of type **DOMAIN_OBJS** from which the named variable list is to be deleted.
name	This is a pointer to the name of the variable list being deleted.

Return Value:

ST_RET	**SD_SUCCESS**.	No error.
	<> 0	Error Code.

Data Structures Used:

See **Module 4 — VMD Control** starting on page 2-10, and **Module 6 — Domain Management** starting on page 2-277 for more information on these data structures.

ms_del_nvlist_obj

Usage: This function deletes a named variable list defined by the **OBJECT_NAME** element **obj** and returns a pointer to it. It differs from the **ms_del_nvlist** function in that a pointer to the **DOMAIN_OBJS** structure is not needed because a call to **ms_find_domain_objs** is made internally. It finds the appropriate domain using the information in the **OBJECT_NAME** structure and then calls the **ms_del_named_vlist** function. This function is used if a pointer to the domain object structure is not known. If more than one **VMD_CTRL** structure is being used locally, be sure to set **m_vmd_select** to point to the appropriate VMD control structure that contains this domain before calling this function.

Function Prototype:

```
ST_RET ms_del_nvlist_obj (OBJECT_NAME *obj, ST_INT chan);
```

Parameters:

`obj` This is a pointer to the object name structure, **OBJECT_NAME**, from which the named variable list is to be deleted.

`chan` This indicates the channel number. This argument is only used if the NamedVariableList defined by **obj** is Application-Association (AA-) specific.

Return Value: `ST_RET` **SD_SUCCESS**. No error.

`<> 0` Error Code.

Data Structures Used:

See **Module 4 — VMD Control** starting on page 2-10, and **Module 6 — Domain Management** starting on page 2-277 for more information on these data structures.

ms_find_named_var

Usage: This support function obtains a pointer to the specific **NAMED_VAR** structure corresponding to a named variable defined in a specified **DOMAIN_OBJS** structure. This is used to obtain the required pointer so that the members of the **NAMED_VAR** structure can be accessed or modified as necessary and appropriate. If more than one **VMD_CTRL** strucutre is being used locally, be sure to set **m_vmd_select** to point to the appropriate VMD control structure that contains this domain before calling this function.

Function Prototype:

```
NAMED_VAR *ms_find_named_var (DOMAIN_OBJS *dom,
                              ST_CHAR *name);
```

Parameters:

dom	This is a pointer to the domain object structure, **DOMAIN_OBJS**, in which the named variable exists.
name	This is a pointer to the name of the variable to find.

Return Value:

NAMED_VAR *	This function returns a pointer to the named variable structure corresponding to the specified variable. This structure is a member of the variable list pointed to by the **var_list** member of the specified **DOMAIN_OBJS** structure. In case of an error, the pointer is set to null and **mms_op_err** is written with the error code.

Data Structures Used:

See **Module 4 — VMD Control** starting on page 2-10, and **Module 6 — Domain Management** starting on page 2-277 for more information on these data structures.

ms_find_named_var_obj

Usage: This support function finds the Named Variable defined by the **OBJECT_NAME** element **obj** and returns a pointer to it. It differs from the **ms_find_named_var** function in that a pointer to the **DOMAIN_OBJS** structure is not needed because a call to **ms_find_domain_objs** is made internally. It finds the appropriate domain using information in the **OBJECT_NAME** structure and then calls **ms_find_named_var**. This function is used if a pointer to the domain object structure is not known. If more than one **VMD_CTRL** structure is being used, be sure to set **m_vmd_select** to point to the appropriate VMD control structure that contains this domain before calling this function.

Function Prototype:

```
NAMED_VAR *ms_find_named_var_obj (OBJECT_NAME *obj,
                                  ST_INT chan);
```

Parameters:

`obj` This pointer to the object name structure, **OBJECT_NAME**, specifies the named variable to find.

`chan` This indicates the channel number to use if the named variable to find is Application-Association (AA-) specific.

Return Value:

`NAMED_VAR *` This function returns a pointer to the named variable structure corresponding to the specified variable. This structure is a member of the variable list pointed to by the **var_list** member of the appropriate **DOMAIN_OBJS** structure. In case of an error, the pointer is set to null and **mms_op_err** is written with the error code.

Data Structures Used:

See **Module 4 — VMD Control** starting on page 2-10, and **Module 6 — Domain Management** starting on page 2-277 for more information on these data structures.

ms_find_nvlist

Usage: This support function obtains a pointer to the specific **NAMED_VAR_LIST** structure corresponding to a named variable list defined in a specified **DOMAIN_OBJS** structure. This is used to obtain the required pointer so that the members of the **NAMED_VAR_LIST** structure can be accessed or modified as necessary and appropriate. If more than one **VMD_CTRL** structure is being used locally, be sure to set **m_vmd_select** to point to the appropriate VMD control structure that contains this domain before calling this function.

Function Prototype:

```
NAMED_VAR_LIST *ms_find_nvlist (DOMAIN_OBJS *dom,
                                ST_CHAR *name);
```

Parameters:

`dom` This is a pointer to the domain object structure, **DOMAIN_OBJS**, in which the named variable exists.

`name` This is a pointer to the named variable list name to find. This is a null-terminated ASCII string.

Return Value:

`NAMED_VAR_LIST *` This function returns a pointer to the named variable list structure corresponding to the specified named variable list. This structure is a member of the named variable list pointed to by the **NAMED_VAR_LIST** member of the specified **DOMAIN_OBJS** structure. In case of an error, the pointer is set to null and **mms_op_err** is written with the error code.

Data Structures Used:

See **Module 4 — VMD Control** starting on page 2-10, and **Module 6 — Domain Management** starting on page 2-277 for more information on these data structures.

ms_find_nvlist_obj

Usage: This support function finds the Named Variable List defined by the **OBJECT_NAME** element **obj** and returns a pointer to it. It differs from the **ms_find_nvlist** function in that a pointer to the **DOMAIN_OBJS** structure is not needed because a call to **ms_find_domain_objs** is made internally. It finds the appropriate domain using information in the **OBJECT_NAME** structure and then calls **ms_find_nvlist**. This function is used if a pointer to the domain object structure is not known. If more than one **VMD_CTRL** structure is being used locally, be sure to set **m_vmd_select** to point to the appropriate VMD control structure that contains this domain before calling this function.

Function Prototype:

```
NAMED_VAR_LIST *ms_find_nvlist_obj (OBJECT_NAME *obj,
                                    ST_INT chan);
```

Parameters:

`obj` This pointer to the object name structure, **OBJECT_NAME**, specifies the named variable list object to find.

`chan` This indicates the channel number to use if the named variable list to find is Application-Association (AA-) specific.

Return Value:

`NAMED_VAR_LIST *` This function returns a pointer to the named variable list structure corresponding to the specified variable. This structure is a member of the named variable list pointed to by the **named_var_list** member of the appropriate **DOMAIN_OBJS** structure. In case of an error, the pointer is set to null and **mms_op_err** is written with the error code.

Data Structures Used:

See **Module 4 — VMD Control** starting on page 2-10, and **Module 6 — Domain Management** starting on page 2-277 for more information on these data structures.

Named Type Manipulation

The functions described on the following pages are called by the user's application program and perform various support functions required by MMS-EASE. They are used to access and update the virtual machine database of named types:

To make use of the Virtual Machine data handling facilities, named data types must be defined. These types are presented to MMS-EASE in the MMS ASN.1 Type Definition PDU format. They are then transformed into an internal runtime format. This internal format is used by MMS-EASE in converting data from an ASN.1-encoded format to local representation, and visa versa. It also performs extensive checking for data type validity.

Standard Types

MMS-EASE has a set of useful types predefined for use by the application program. These **S**ISCO **T**ypes (ST_) are defined in terms of the MMS-EASE Global type definitions. See **Volume 1 — Appendix E** and the **glbtypes.h** for more information.

Type Name	Description	MMS-EASE Global Type
Integer8	Signed Character	ST_INT8
Integer16	Signed Short	ST_INT16
Integer32	Signed Long	ST_INT32
Integer64	Signed Long	ST_INT64
Unsigned8	Unsigned Character	ST_UINT8
Unsigned16	Unsigned Short	ST_UINT16
Unsigned32	Unsigned Long	ST_UINT32
Unsigned64	Unsigned Long	ST_UINT64
Boolean	Unsigned Character	ST_BOOLEAN
Float		ST_FLOAT
Double		ST_DOUBLE
Void		ST_VOID
String8		ST_CHAR[9]
String16		ST_CHAR[17]
String32		ST_CHAR[33]
String64		ST_CHAR[65]
String128		ST_CHAR[129]
String256		ST_CHAR[257]
Integer16	General Purpose Return Code	ST_RET

TABLE 1: MMS-EASE Standard Types

These types can be added into the MMS-EASE environment by calling the support function, **`ms_add_std_types`**. See page 2-83 for more information on this function. They can then be deleted or used as type definitions in subsequent variable accesses. Note that types defined for strings contain an extra byte at the end of them. This is so the standard C language libraries can be used that implement null-terminated strings for string manipulation.

NOTE: You may wish to use more complex types than what is provided by the MMS-EASE Standard Types. The **`ms_add_named_type`** function is supplied for this purpose, and allows any type found as data to be transferred as networked data in a C program to be modeled as MMS data.

To know if a structure in a C program will be accessed, the following steps can be taken.

1. Print the size of a specified structure by using the C statements:
   ```
   printf(sizeof(struct structure_name));
   ```
2. Look at the blocked length of the MMS-EASE type.
3. Compare results of these two steps, they should match. If they do not match, then the structure may be rearranged to change the padding inserted by the compiler. Another alternative is to adjust the structure member offsets using the **`m_def_data_algn_table`**.

Named Type Manipulation Functions

ms_add_named_type

Usage: This support function adds a type definition into the type definition table for a specific domain or VMD. This type definition is placed somewhere in the **type_list** member of the **DOMAIN_OBJS** structure. After the function is called, the (***write_ind_fun**), (***read_ind_fun**) and other appropriate members of the **type** structure (of type **NAMED_TYPE**) can be filled out if necessary. The **ms_mk_asn1_type** function can be used (with the MMS-EASE Type Description Language, as explained on page 2-91), to build the ASN.1 type definition needed by this function. If more than one **VMD_CTRL** structure is being used locally, be sure to set **m_vmd_select** to point to the appropriate VMD control structure containing this domain before calling this function.

Function Prototype:

```
struct named_type *ms_add_named_type (DOMAIN_OBJS *dom,
                                      ST_CHAR *name,
                                      ST_UCHAR *asn1,
                                      ST_INT asn1len);
```

Parameters:

dom	This is a pointer to the domain object structure, **DOMAIN_OBJS**, into which the type definition is to be added.
name	This is a pointer to the name of the type being created.
asn1	This is a pointer to ASN.1 encoded type definition. This ASN.1 buffer must be static in memory. See **Volume 1 — Appendix I** for more information on creating type specifications.
asn1len	This is the length of the ASN.1 encoded type definition.

Return Value:

NAMED_TYPE *	This pointer to the type definition structure is used for the type just added. This structure is a member of the type list pointed to by the **type_list** member of the specified **DOMAIN_OBJS** structure. In case of an error, the pointer is set to null and **mms_op_err** is written with the error code.

Data Structures Used:

See **Module 4 — VMD Control** starting on page 2-10, and **Module 6 — Domain Management** starting on page 2-277 for more information on these data structures.

ms_add_std_types

Usage: This function is used to add all the MMS-EASE standard variable type definitions into the type definition database for a specific domain or VMD. It should be called after **strt_MMS** is called, and after the specified domain, and VMD have been created. These type definitions are placed somewhere in the **type_list** member of the specified **DOMAIN_OBJS** structure. If this function is not called or if other type definitions are needed, the user program is responsible for entering these by calling **ms_add_named_type**. See **Volume 1 — Appendix I** for information on type specification. If more than one **VMD_CTRL** structure is being used locally, be sure to set **m_vmd_select** to point to the appropriate VMD control structure containing this domain before calling this function.

Function Prototype:

```
ST_RET ms_add_std_types (DOMAIN_OBJS *dom);
```

Parameters:

`dom` This is a pointer to the domain object structure of type **DOMAIN_OBJS** into which the type definitions are to be added.

Return Value: `ST_RET`

SD_SUCCESS. No Error.

`> 0` Error Code.

`< 0` Types already present in this domain

Data Structures Used:

See **Module 4 — VMD Control** starting on page 2-10, and **Module 6 — Domain Management** starting on page 2-277 for more information on these data structures.

ms_get_blocked_len

Usage: This function is used to calculate the number of bytes of memory required by a derived MMS type represented as an array of **RUNTIME_TYPE** structures.

Function Prototype:
```
ST_INT ms_get_blocked_length (RUNTIME_TYPE *rt,      ST_INT
                                                num_rt);
```

Parameters:

rt	This is a pointer to an array of structures of type **RUNTIME_TYPE** that represents the derived MMS type.
num_rt	This is the number of elements in the **rt** array.

Return Value: ST_INT — This is the number of bytes including any pad bytes that are required to store a C language data type in memory.

ms_del_all_named_types

Usage: This support function deletes ALL defined named types from the specified **DOMAIN_OBJS** structure. If more than one **VMD_CTRL** structure is being used locally, be sure to set **m_vmd_select** to point to the appropriate VMD control structure containing this domain before calling this function.

Function Prototype:

```
ST_VOID  ms_del_all_named_types (DOMAIN_OBJS *dom_objs);
```

Parameters:

`dom_objs` This pointer to the domain object structure of type **DOMAIN_OBJS** contains the set of named types to be deleted.

Return Value: `ST_VOID` (ignored)

NOTE: If types are deleted that are still referenced by variables, they will be marked as logically deleted. They will not be physically deleted until all the variables referencing them are deleted.

Data Structures Used:

See **Module 4 — VMD Control** starting on page 2-10, and **Module 6 — Domain Management** starting on page 2-277 for more information on these data structures.

ms_del_named_type

Usage: This function deletes a defined named type from the specified **DOMAIN_OBJS** structure. If more than one **VMD_CTRL** structure is being used locally, be sure to set **m_vmd_select** to point to the appropriate VMD control structure containing this domain before to calling this function.

This function differs from the **ms_del_type** function in that a pointer to the **NAMED_TYPE** structure is not needed because a call to **ms_find_named_type_objs** is made internally. It finds the appropriate type using the information in the **OBJECT_NAME** structure and then calls the **ms_del_type** function. This function is used if a pointer to the named type structure is not known.

Function Prototype:

```
ST_RET ms_del_named_type (DOMAIN_OBJS *dom,
                          ST_CHAR *name);
```

Parameters:

`dom` This is a pointer to the domain object structure of type **DOMAIN_OBJS** from which the named type is to be deleted.

`name` This is a pointer to the name of the type being deleted.

Return Value: `ST_RET`

SD_SUCCESS. No error.

`<> 0` Error Code.

NOTE: If types are deleted that are still referenced by variables, they will be marked as logically deleted. They will not be physically deleted until all the variables referencing them are deleted.

Data Structures Used:

See **Module 4 — VMD Control** starting on page 2-10, and **Module 6 — Domain Management** starting on page 2-277 for more information on these data structures.

ms_del_named_type_obj

Usage: This function deletes a named type defined by the **OBJECT_NAME** element **obj**. It differs from the **ms_del_named_type** function in that a pointer to the **DOMAIN_OBJS** structure is not needed because a call to **ms_find_domain_objs** is made internally. It finds the appropriate domain using the information in the **OBJECT_NAME** structure and then calls the **ms_del_named_type** function. This function is used if a pointer to the domain object structure is not known. If more than one **VMD_CTRL** structure is being used locally, be sure to set **m_vmd_select** to point to the appropriate VMD control structure that contains this domain before calling this function.

Function Prototype:

```
ST_RET ms_del_named_type_obj (OBJECT_NAME *obj,
                              ST_INT chan);
```

Parameters:

`obj` This is the pointer to the object name structure of type **OBJECT_NAME** from which the type variable is to be deleted.

`chan` This indicates the channel number. This argument is only used if the Named Type defined by **obj** is Application-Association (AA-) specific.

Return Value: `ST_RET` **SD_SUCCESS**. No error.

`<> 0` Error Code.

NOTE: If types are deleted that are still referenced by variables, they will be marked as logically deleted. They will not be physically deleted until all the variables referencing them are deleted.

Data Structures Used:

See **Module 4 — VMD Control** starting on page 2-10, and **Module 6 — Domain Management** starting on page 2-277 for more information on these data structures.

ms_del_type

Usage: This function deletes a named type from the specified **DOMAIN_OBJS** structure. It requires that a pointer to this named type must be used. If more than one **VMD_CTRL** structure is being used locally, be sure to set **m_vmd_select** to point to the appropriate VMD control structure containing this domain before to calling this function.

Function Prototype: `ST_RET ms_del_type (DOMAIN_OBJS *dom, NAMED_TYPE *name);`

Parameters:

`dom` This is a pointer to the domain object structure of type **DOMAIN_OBJS** from which the named type is to be deleted.

`name` This structure of type **NAMED_TYPE** points to the defined type to be deleted.

Return Value: `ST_RET` **SD_SUCCESS**. No error.

`<> 0` Error Code.

NOTE: If types are deleted that are still referenced by variables, they will be marked as logically deleted. They will not be physically deleted until all the variables referencing them are deleted.

Data Structures Used:

See **Module 4 — VMD Control** starting on page 2-10, and **Module 6 — Domain Management** starting on page 2-277 for more information on these data structures.

ms_find_named_type

Usage: This support function obtains a pointer to the specific **NAMED_TYPE** structure corresponding to a named type defined in a specified **DOMAIN_OBJS** structure. This function is used to obtain the required pointer so that the members of the **NAMED_TYPE** structure can be accessed or modified as necessary. If more than one **VMD_CTRL** structure is being used locally, be sure to set **m_vmd_select** to point to the appropriate VMD control structure that contains this domain before calling this function.

Function Prototype:

```
NAMED_TYPE *ms_find_named_type (DOMAIN_OBJS *dom,
                                ST_CHAR *name);
```

Parameters:

`dom` This is a pointer to the domain object structure of type **DOMAIN_OBJS** in which the named type exists.

`name` This is a pointer to the name of the type to find.

Return Value:

`NAMED_TYPE *` This pointer to the type definition structure corresponds to the specified type. This structure is a member of the type list pointed to by the **type_list** member of the appropriate **DOMAIN_OBJS** structure. In case of an error, the pointer is set to null and **mms_op_err** is written with the error code.

Data Structures Used:

See **Module 4 — VMD Control** starting on page 2-10, and **Module 6 — Domain Management** starting on page 2-277 for more information on these data structures.

ms_find_named_type_obj

Usage: This support function finds the Named Type defined by the **OBJECT_NAME** element **obj** and returns a pointer to it. It differs from the **ms_find_named_type** function in that a pointer to the **DOMAIN_OBJS** structure is not needed because a call to **ms_find_domain_objs** is made internally. It finds the appropriate domain using information in the **OBJECT_NAME** structure and then calls **ms_find_named_type**. This fun ction is used if a pointer to the domain object structure is not known. If more than one **VMD_CTRL** structure is being used locally, be sure to set **m_vmd_select** to point to the appropriate VMD control structure that contains this domain before calling this function.

Function Prototype:

```
NAMED_TYPE *ms_find_named_type_obj (OBJECT_NAME *obj,
                                    ST_INT chan);
```

Parameters:

obj	This pointer to the object name structure, **OBJECT_NAME**, specifies the named type to find.
chan	This indicates the channel number to use if the named variable to find is Application-Association (AA-) specific.

Return Value:

NAMED_TYPE *	This function returns a pointer to the named variable structure corresponding to the specified variable. This structure is a member of the variable list pointed to by the **var_list** member of the appropriate **DOMAIN_OBJS** structure. In case of an error, the pointer is set to null and **mms_op_err** is written with the error code.

Data Structures Used:

See **Module 4 — VMD Control** starting on page 2-10, and **Module 6 — Domain Management** starting on page 2-277 for more information on these data structures.

MMS-EASE Type Description Language

To create an ASN.1-encoded type specification, you would first create an ASCII string that represents that type using the MMS-EASE **T**ype **D**escription **L**anguage (TDL). TDL allows describing variable types in a much easier-to-understand manner than the ASN.1-encoded type specification. The **`ms_mk_asn1_type`** function is provided for creating the appropriate ASN.1-encoded type specification from a TDL string. The user application can then use the ASN.1-encoded type specification in a call to the **`ms_add_named_type`** function (see page 2-64), or by filling out the **`VAR_ACC_TSPEC`** structure (see page 2-91).

TDL consists of two types of elements:

1. Predefined names used to describe simple types that will be combined to form a complex type.
2. Structure control marks used to specify the start and end of items such as structures, arrays, lengths.

Simple Type Names

The following is a description of the simple type names used by TDL and their corresponding C language representation in terms of the MMS-EASE global type definitions.

BCD This type is encoded as a MMS signed integer where the value is dependent on the length **x** of the BCD type. **x** represents the number of 4 bit nibbles in the type. Each place specified by **x** may hold a value [0..9]. MMS-EASE only supports BCD types where **x** is [1..8]. The C language representation of BCD is a signed integer. The size of the integer used to hold the type varies according to **x**. A **`ST_INT8`** should be used when **x** is [3..4]. The **`ST_INT32`** integer is used when **x** is [5..8]. The application is responsible for converting any native BCD data to its signed integer equivalent before sending the value. Similarly, the signed integer must be converted back to native BCD.

Example: 10 BCD ⇨ `0x0010`
convert to `0x000A` before sending

Bool This type is encoded as a MMS Boolean variable. The value of variables of this type take on only two values: **`SD_TRUE`** (<> 0) or **`SD_FALSE`** (= 0). The SISCO macro for the C language representation of **Bool** is **`ST_BOOLEAN`**.

Byte This type is encoded as a MMS signed integer one byte in length where the value must be between -128 and +127. The SISCO macro for the C language representation of **Byte** is **`ST_INT8`**. Do not use this type of variable to store ASCII; use one of the string types instead.

Short This type is encoded as a MMS signed integer two bytes in length where the value must be between -32,768 and +32,767. The SISCO macro for the C language representation of **Short** is **`ST_INT16`**.

Long This type is encoded as a MMS signed integer four bytes in length where the value must be between -2^{31} and $+2^{31}-1$. The SISCO macro for the C language representation of **Long** is **`ST_INT32`**.

Int64 This type is encoded as a MMS signed integer eight bytes in length where the value must be between -2^{63} and $+2^{63}-1$. The SISCO macro for the C language representation of **Uint64** is **`ST_INT64`**.

Ubyte This type is encoded as a MMS unsigned integer one byte in length where the value must be between 0 and 255. The SISCO macro for the C language representation of **Ubyte** is **`ST_UINT8`**. Do not use this type of variable to store ASCII; use one of the string types instead.

Ushort This type is encoded as a MMS unsigned integer two bytes in length where the value must be between 0 and 65,535. The SISCO macro for the C language representation of **Ushort** is **`ST_UNT16`**.

Ulong This type is encoded as a MMS unsigned integer four bytes in length where the value must be between 0 and $+2^{32}-1$. The SISCO macro for the C language representation of **Ulong** is **`ST_UINT32`**.

Uint64 This type is encoded as a MMS unsigned integer eight bytes in length where the value must be between 0 and 264 -1. The SISCO macro for the C language representation of **Uint64** is **`ST_UINT64`**.

Float This type is encoded as a single precision MMS floating point. The mantissa and exponent lengths are properly encoded to match the local format. The SISCO macro for the C language representation of **Float** is **`ST_FLOAT`**.

Double This type is encoded as a double precision MMS floating point. The mantissa and exponent lengths are properly encoded to match the local format. The SISCO macro for the C language representation of **Double** is **`ST_DOUBLE`**.

Gtime This type is encoded as MMS Generalized Time(Gtime). The C representation of Generalized Time is a **`time_t`** structure. This is an ANSI C typedef and is included in a header file supplied by the authors of the compiler. The value of the **`time_t`** variable in your application is treated as the number of seconds from midnight starting January 1, 1970. The value will only be encoded and decoded correctly if the time is greater than midnight starting January 1, 1984. This is because (time as described in the MMS spec) is relative to January 1, 1984. C Language implementations however, usually only have time functions relative to January 1, 1970.

Btime4 This type is encoded as Binary TimeOfDay with no days. The SISCO macro for the C language representation of Btime4 is **`ST_INT32`**. This value represents the number of milliseconds since midnight of the current day.

Btime6 This type is encoded as BinaryTimeOfDay with days relative to January 1, 1984. The SISCO macro for the C language representation of Btime6 is a structure containing two consecutive **`ST_INT32`**. The value contained in the first **`ST_INT32`** represents the number of milliseconds since midnight of the current day. The value contained in the second **`ST_INT32`** represents the number of days relative to January 1 1984. This is because time (as described in the MMS spec) is relative to January 1, 1984. C Language implementations however, usually only have time functions relative to January 1, 1970.

VstringXX This type is encoded as a MMS visible string of a variable length not to exceed XX bytes. Variables of this type should be used to store variable length VisibleStrings. Only the 7-bit ASCII characters minus the control characters (31 < char < 127) can be represented by a VisibleString. For instance, MMS Object Names are encoded as VisibleStrings but can only contain the **$** and **_** punctuation marks, and the alphanumeric characters. If you need to send non VisibleString data, use the octet string (OstringXX) instead. The SISCO macro for the C language representation of **VstringXX** is **`ST_CHAR`**[XX+1], where XX is the number of characters in the string. The extra byte in the C language representation is used to store the null used by the C language. The null is not sent on the wire. The length of this type of variable, specified by the XX, is the maximum length that the variable can be. MMS-EASE only sends or receives data up to a null or XX bytes for variables of this type. For example, “Vstring24” specifies a VisibleString with no more than 24 characters.

FstringXX This type is encoded as a MMS visible string of a fixed length of XX bytes. Variables of this type should be used to store fixed length VisibleStrings. Only the 7-bit ASCII characters minus the control characters (31 < char < 127) can be represented by a fixed length VisibleString. If you need to send non VisibleString data use the octet string type (OstringXX) instead. The C language representation of **FstringXX** is **`ST_CHAR`**[XX+1], where XX is the number of characters in the string. The extra byte in the C language representation is used to store the null used by the C language. The null is not sent on the wire. The length of this type of variable as specified by the XX is the actual length that will be sent on the wire. MMS-EASE sends all bytes specified by the length. This is so if the actual data does not occupy the entire string, the remainder of the string will have to be padded with spaces so that the entire length is XX bytes. For example, Fstring16 specifies a fixed length VisibleString consisting of exactly 16 characters.

OstringXX This type is encoded as a MMS OctetString of a fixed length of XX bytes. Variables of this type should be used to store binary data or character data that does not conform to the limitations specified for VisibleStrings. Each individual character of an OctetString can take on any value between 0 and 255. The SISCO macro for the C language representation of **OstringXX** is **ST_UCHAR**[XX], where XX is the number of bytes of data in the string. Note that there is no extra byte for the null because a null can be a valid member of an OctetString. The length of this type of variable as specified by the XX is the actual length that will be sent on the wire. For example, Ostring256 specifies a data stream of exactly 256 bytes.

OVstringXX This type is encoded as a MMS OctetString of a variable length not to exceed XX bytes.Variables of this type should be used to store binary data or character data that does not conform to the limitations specified for VisibleStrings. Each individual character of an OctetString can take on any value between 0 and 255. The length of this type of variable as specified by the XX is the maximum length that will be sent on the wire. For example, OVstring256 specifies a data stream of less than or equal to 256 bytes. The SISCO structure for the C language representation of **OVstringXX** is:

```
struct  ovstring {
   ST_INT16  len;
   ST_UCHAR  data[XX];
};
```

The name and placement of the structure declaration is up to the application. **len** is the number of bytes of data in the string not to exceed XX. Note that there is no extra byte for the null because a null can be a valid member of an OctetString.

BstringXX This type is encoded as a MMS BitString of a fixed length of XX bits. The SISCO macro for the C language representation of **BstringXX** is an **ST_UCHAR** array where each individual byte of the array contains no more than 8 bits. The bit numbering within each byte starts with the most significant bits having a smaller bit number than the least significant bits of the byte. Therefore, if the bitstring length, specified by XX, is not a multiple of 8, MMS-EASE only uses the necessary number of most significant bits of the last byte needed to complete the bit string. The least significant bits of the last byte will be ignored.

BVstringXX This type is encoded as a MMS BitString of a variable length of not to exceed XX bits . The bit numbering within each byte starts with the most significant bits having a smaller bit number than the least significant bits of the byte. Therefore, if the bitstring length, specified by XX, is not a multiple of 8, MMS-EASE only uses the necessary number of most significant bits of the last byte needed to complete the bit string. The least significant bits of the last byte will be ignored. The SISCO structure for the C language representation of **BstringXX** is shown below:

```
struct  bvstring {
   ST_INT16  len;
   ST_UCHAR  data[YY];
   };
```

The name and placement of the structure declaration is up to the application. **len** is the number of bits of data in the string not to exceed XX. YY is the number of bytes in the array equal to (XX+7)/8.

Structure Control

MMS-EASE TDL uses punctuation marks and other pre-defined sequences of characters to signal the beginning and end of structures and arrays. They provide other type related information such as pre-named types, and VMD names. The following is a description of the various structure control character sequences, and what they mean to the TDL:

{ } The pillow marks are used to signal the beginning "**{**" and the end "**}**" of complex structure definitions.

[ ] The brace marks signal the beginning "**[**" and the end "**]**" of array definitions. Immediately following the start of an array symbol "**[**", there should be either a "**p**" as described below, or a number indicating the number of elements in the array.

p This symbol immediately following the start of an array or structure indicates that all elements within the array or structure are to be packed. Note that MMS-EASE defaults to non-packed variables suitable for most applications. Non-packed means that all elements of a data structure will be placed on word, not byte boundaries. All the MMS-EASE defined data structures are not packed, and must remain on word boundaries. Only user defined named types and the corresponding named variables can be packed.

: A colon is used to separate various fields within a type specification such as the number of elements in an array from the type name for the members of the array, and the domain name from a pre-existing type name.

( ) Parenthesis are used to signal the start "**(**" and end "**)**" of the name of an individual element of a structure. All element names must be MMS Identifiers. These must be VisibleStrings no longer than 32 characters that exist only of numbers (0-9), upper and lower case letters (A-Z, a-z), the _ and **$** marks.

< > The right and left angles are used to signal the start "<" and end ">" of references to pre-named types. This allows you to cross-reference pre-existing named types already placed in the MMS-EASE database when building subsequent type definitions.

@ The "at" (**@**) symbol is used to reference pre-existing named types that are either VMD specific (@VMD) or Application-Association specific (@AA).

Examples

Several examples are provided of how to build complex type definitions using the TDL.

Example #1:

Create the ASN.1 Type Definition for the following structure:

```
struct
  {
  ST_CHAR  name[33];          /*  Visible string containing name     */
  ST_INT16 tag_value;       /*  A short signed integer              */
  ST_INT32 time_array[32];  /*  An array of 1                       */
  } test1;
```

1. The TDL descriptor for this type is:

   { Vstring32, Short, [32:Long] }

2. If the individual element names were added into the type definition, the TDL descriptor becomes:

   ```
   { (name) Vstring32, (tag_value) Short, (time_array) [32:Long]}
   ```

3. To create the ASN.1 type definition, without the element names, string #1 is passed to the **`ms_mk_asn1_type`** function. Then, it is added into the existing domain named "domain1." A 1K buffer will be allocated for holding the type definition.

```
ST_CHAR   *buf_ptr;
ST_CHAR   *type_ptr;
ST_CHAR   *type_string = "{Vstring33, Short, [32:Gtime] }\0"
ST_INT16 type_len = 1024; /* initialize to maximum size of buffer     */
buf_ptr = chk_malloc(1024)
type_ptr = ms_mk_asn1_type (&type_len,buf_ptr,type_string);
```

After the call to **ms_mk_asn1_type** is made and a valid pointer is returned, **type_ptr** points to the beginning of the ASN.1 type definition. The length will be written into**type_len**. This ASN.1 type definition can be used to add a named type corresponding to this definition into the virtual machine database using a call to **ms_add_named_type**. See the description of the **ms_mk_asn1_type** function on page 2-76 for more information.

Example #2:

Create the TDL descriptor for the following array of structures.

```
struct
  {
  ST_CHAR    mask[7];           /* 56 bit bitstring */
  struct     test1  sample1;
  struct     test2  sample2;
  struct     test3  sample3;
  ST_FLOAT   value;
  }  test4[16];
```

Assume:

1. That the type definition for test1 has already been added into the MMS-EASE virtual machine database as a VMD specific named type.
2. That the type definition for test2 has already been added into the MMS-EASE virtual machine database as a domain specific named type.
3. That the type definition for test3 has already been added into the MMS-EASE database as an Application-Association (AA-)specific named type.

This results in the following TDL descriptor:

```
[16:{Bstring56,<@VMD:test1>,<domain1:test2>,<@AA:test3>,Float}]
```

If adding names to the elements, the TDL Descriptor becomes:

```
[16:{(mask)Bstring56,(sample1)<@VMD:test1>,(sample2)<domain1:test2>,
   (sample3)<@AA:test3>,(value)Float}]
```

The use of spaces is optional. They may be included to make the TDL descriptor easier to read.

NOTES:

1. Care must be taken when using the Btime4 and Btime6 types. These types only specify time with respect to the local time zone, there may be problems if the data crosses a time zone. Also, Btime4 does not contain date information. This may add additional confusion. Although the Gtime type specifies time with respect to Greenwich Mean Time, it requires that your computer be set up with the proper time and time zone information in order for the operating system to supply you with time properly for Gtime. Remember, these types only exist on the network. The time format used by your application program is that of the C language for your system. MMS-EASE takes care of converting between the C time and the Gtime, Btime4, or Btime6.
2. Do not nest structures within arrays, arrays within structures, arrays within arrays, structures within structures more than 10 deep.
3. If an error is detected while attempting to convert the TDL descriptor into ASN.1, a null pointer is returned by **ms_mk_asn1_type**. An error code (as described in **Volume 1 — Appendix A**) is placed in **mms_op_err**.

TDL Manipulation Function

ms_mk_asn1_type

Usage: This support function creates the ASN.1 Type Definition of a type using the MMS-EASE Type Description Language (TDL) as an input. The ASN.1 that is output from this function can be used as an input to the **ms_add_named_type** function. This can be used to add a named type definition corresponding to the TDL descriptor supplied to this function into the MMS-EASE database. See **Volume 1 — Appendix I** for more information on constructing TDL descriptors in order to describe complex types to the MMS-EASE environment. The actual length of the ASN.1 Type Definition is written into the location pointed to by **asn1_len**.

Function Prototype:

```
ST_UCHAR *ms_mk_asn1_type (ST_INT *asn1_len,
                           ST_UCHAR *asn1_buf,
                           ST_CHAR *tdl);
```

Parameters:

`asn1_len` This argument is used for both input and output to the function. On input, the value it points to will specify the length of the ASN.1 buffer to which the **asn1_buf** argument points. On output, the value it points to is changed to be the actual length of the ASN.1 type definition pointed to by the return value of this function.

`asn1_buf` This is a pointer to the beginning of the ASN.1 buffer where the ASN.1 Type Definition is to be built. The Type Definition is built in the buffer starting from the back of this buffer. The return value of this function points to a location somewhere in the middle of this buffer.

`tdl` This pointer to the TDL descriptor string specifies the type to be converted into ASN.1. See **Volume 1 — Appendix I** for more information on type specification.

Return Value:

`UCHAR *` This is a pointer to the beginning of the ASN.1 Type Definition. The variable pointed to by **asn1_len** is written with the actual length of the ASN.1 Type Definition. In case of an error, the pointer is set to null and **mms_op_err** is written with the error code.

General Data Alignment

Previously, the MMS-EASE Virtual Machine data handling system assumed that all data is packed in memory. This can cause problems in mapping network data onto local data, due to alignment options, requirements, and other system-specific characteristics. The 'packed' local representation can sometimes be inconvenient for the application programmer, especially in the cases where the compiler is inflexible in structure alignment options.

Described below is a modified data handling system now in place. This system allows the application to always use natural access to structure members (no shifts, masks, etc.), and will support the use of any desired data alignment scheme. The default data alignment is the same as in previous versions of MMS-EASE.

Data alignment can be normally be adequately controlled by setting a global data alignment control table pointer to achieve the desired alignment result. There are two standard alignment control tables included in the MMS-EASE libraries:

- For packed data alignment (no pad bytes used), the application does not need to take any action, as this is the default:
  `*m_data_algn_tbl = m_packed_data_algn_tbl`
- To allow 'natural' access to structure and array components, use the following:
  `*m_data_algn_tbl = m_def_data_algn_tbl`

Most new applications should select the **`m_def_data_algn_tbl`** as the alignment table, as this will provide more natural access to structure elements and will promote portability. **Advanced Data Alignment - MMS-EASE Runtime Type**

The MMS-EASE Virtual Machine data handling makes use of a "runtime type" mechanism. A runtime type is a table of **`RUNTIME_TYPE`** structures that describes a data type. A runtime type is usually created when the Virtual Machine decodes an ASN.1 Type Specification using the **`ms_asn1_to_runtime`** support function. This can occur, for example, after a named type is added using **`ms_added_named_type`**.

One aspect of a runtime type that may be important is the aspect that affects the data component alignment. Each element of a runtime type contains an **`el_size`** component. This tells the data conversion tools how much memory that element consumes including any trailing pad bytes. The **`el_size`** component is set during the ASN.1 to Runtime process. The way this component is calculated can be controlled by the application.

The Runtime element size calculation function takes a table of DEFAULTs that define the alignment requirements for each primitive data type plus array and structure start and end elements. The array is selected using the global pointer: **`*m_data_algn_tbl`**. By default, this pointer selects the table: **`m_packed_data_algn_tbl [NUM_ALGN_TYPES]`**. That table assumes that all data elements are packed in memory. An additional sample data alignment table is included that reflects the default alignment of the target compiler: **`m_def_data_algn_tbl [NUM_ALGN_TYPES]`**. Most new applications should select the **`m_def_data_algn_tbl`** as the alignment table since this will provide a more natural access to structure elements and is portable across platforms. Assigning **`m_data_algn_table = m_def_data_algn_tbl`** should take place in the application sometime after **`strt_MMS`** has been called but before the first NamedType is added.

However, it is possible and easy to define a custom data alignment table for special situations. When a runtime type element is encountered, the **`el_tag`** component of the **`RUNTIME_TYPE`** structure is used to index into the current **`m_data_algn_tbl`**, and an "alignment mask" for the data type is selected. The "alignment mask" is then used to determine which bits in the offset from the start of the data element CANNOT be set. For example, an "alignment mask" of `0x0000` indicates that the data element can be on any address (no restrictions). A value of `0x0001` indicates that the low bit of the offset cannot be set and causes the value to be even address aligned. A value of `0x0003` will force the data alignment to be four byte aligned.

Below are shown the index definitions into the alignment table:

```
#define ARRSTRT_ALGN          0
#define ARREND_ALGN           1
#define STRSTRT_ALGN          2
#define STREND_ALGN           3
#define INT8_ALGN             4
#define INT16_ALGN            5
#define INT32_ALGN            6
#define FLOAT_ALGN            7
#define DOUBLE_ALGN           8
#define OCT_ALGN              9
#define BOOL_ALGN             10
#define BCD1_ALGN             11
#define BCD2_ALGN             12
#define BCD4_ALGN             13
#define BIT_ALGN              14
#define VIS_ALGN              15

#define NUM_ALGN_TYPES        16

extern ST_INT *m_data_algn_tbl;
extern ST_INT m_packed_data_algn_tbl[NUM_ALGN_TYPES];
extern ST_INT m_def_data_algn_tbl[NUM_ALGN_TYPES];
extern ST_INT *m_data_algn_tbl = m_packed_data_algn_tbl;
```

A sample table for handling Microsoft's default alignment (align on smaller of word or data type) is shown below.

```
ST_INT msoft_word_algn_tbl[NUM_ALGN_TYPES] =
{
0x0000,   /* ARRSTRT_ALGN     00  */
0x0000,   /* ARREND_ALGN      01  */
0x0000,   /* STRSTRT_ALGN     02  */
0x0000,   /* STREND_ALGN      03  */
0x0000,   /* INT8_ALGN        04  */
0x0001,   /* INT16_ALGN       05  */
0x0001,   /* INT32_ALGN       06  */
0x0001,   /* FLOAT_ALGN       07  */
0x0001,   /* DOUBLE_ALGN      08  */
0x0000,   /* OCT_ALGN         09  */
0x0000,   /* BOOL_ALGN        10  */
0x0000,   /* BCD1_ALGN        11  */
0x0000,   /* BCD2_ALGN        12  */
0x0001,   /* BCD4_ALGN        13  */
0x0000,   /* BIT_ALGN         14  */
0x0000    /* VIS_ALGN         15  */
};
```

Runtime Type Data Structure

This is the structure used by MMS-EASE to keep track of a runtime type. To convert from runtime types to ASN.1, use the **ms_runtime_to_asn1** support function.

```
struct runtime_type
  {
  ST_UCHAR el_tag;
  ST_INT   el_size;
  ST_INT   offset_to_last;
  union
    {
    struct
      {
      ST_INT el_len;
      } p;
    struct
      {
      ST_BOOLEAN  packd;
      ST_INT      num_rt_blks;
      } str;
    struct
      {
      ST_INT      num_elmnts;
      ST_INT      num_rt_blks;
      ST_BOOLEAN  packd;
      ST_INT      loops;
      } arr;
    ST_CHAR name[MAX_IDENT_LEN+1];
    } u;
  };
typedef struct runtime_type RUNTIME_TYPE;
```

Fields:

`el_tag` This indicates the ID tag expected for this element or it can indicate special values such as:

RT_ARR_START	1	Start of a runtime array
RT_STR_START	2	Start of a runtime structure
RT_BOOL	3	Runtime Boolean
RT_BIT_STRING	4	Runtime Bit String
RT_INTEGER	5	Runtime Integer (8-16-32)
RT_UNSIGNED	6	Runtime Unsigned Integer (8-16-32)
RT_FLOATING_POINT	7	Runtime Floating Point
RT_OCTET_STRING	9	Runtime Octet String
RT_VISIBLE_STRING	10	Runtime Visible String
RT_GENERAL_TIME	11	Runtime Generalized Time
RT_BINARY_TIME	12	Runtime Binary Time
RT_BCD	13	Runtime Binary Coded Decimal
RT_BOOLEANARRAY	14	Runtime Boolean Array
RT_ARR_END	15	End of a runtime array
RT_STR_END	16	End of a runtime structure
RT_NAMED_COMP	17	Runtime Named Component
Special values:		
0x80		component name **rt_block**
0x81		end of array
0x82		end of structure

`el_size` This indicates the number of bytes occupied by this element.

`offset_to_last` This indicates the offset in bytes from the start of the data to the last element (first runtime type only).

`el_len` This indicates the precision, in bytes. This is true for all types except Bit Strings.

Structure

`packd` This flag indicates whether the runtime type structure is to be packed.

SD_TRUE. Structure should be packed.

SD_FALSE.Structure should not be packed.

`num_rt_blks` The number of runtime blocks needed to get to either the start or the end element of the structure.

Array

`num_elmnts` The number of array limits (index limit).

`num_rt_blks` The runtime blocks needed to get to either the start or the end element of the array.

`packd` This flag indicates whether the runtime type array is to be packed.

SD_TRUE. Array should be packed.

SD_FALSE. Array should not be packed.

`loops` This parameter is used in runtime conversions.

`name` This indicates a component name for the element.

User Defined Runtime Type Processing

u_rt_type_process

Usage: When a runtime type is created, the runtime type table is post-processed and the size of each element in memory is calculated. After the post-processing, a user function pointer is invoked and is passed the runtime table. This function can do whatever is necessary including further processing or logging.

Function Prototype:

```
extern ST_VOID (*u_rt_type_process) (RUNTIME_TYPE *rt,
                                     ST_INT num_rt);
```

Parameters:

`rt` This pointer to the runtime type definition of type **`RUNTIME_TYPE`** contains the runtime type information corresponding to the data to be converted.

`num_rt` This indicates the number of runtime type elements. The size of these elements can be modified by supplying a new alignment control table. See page 2-97 for more information.

Return Value: `ST_VOID` (ignored)

NOTE: Below is a sample of how the runtime type can be ensured to always be correct, regardless of the alignment used by the compiler. Note that this approach is useful only when the type is known before being processed such as for local data types. This function is used as the runtime process function of the ASN.1 to Runtime operation for the following structure:

```
struct str2
   {
   ST_BOOLEAN  b;
   ST_INT32    l;
   ST_INT      s;
   };

ST_VOID str2_rt_process_fun (struct runtime_type *rt, ST_INT num_rt)
   {
   struct str2 s;
   ST_INT size_above;

      assert (num_rt = 5);                      /* Better be 5!               */

      /* Note that this is used only with the non-AA support data handling*/
      /* functions. Also this particular method includes pad for the last */
      /* element, whereas the standard mechanisms leave off the final pad */

      rt->offset_to_last = sizeof (s);

      rt->el_size = 0;                   /* Structure start      */

      /* This element's size is the difference between it's address & the */
      /* address of the component below.                                  */
      rt++;
      rt->el_size = (ST_CHAR *)&s.l - (ST_CHAR *)&s.b; /* Boolean       */
```

. . . Continued on following page . . .

u_rt_type_process . . . cont'd from preceding page . . .

Notes:

```
    /* This element's size is the difference between it's address & the */
    /* address of the component below.                                  */
    rt++;
    rt->el_size = (ST_CHAR *)&s.s - (ST_CHAR *)&s.l; /* Long           */

    /* Since this is the last element, we need to include the pad. To do*/
    /* this, use the following formula -                                */
    /*  el_size = total_size - size_above,where size_above is the number*/
    /* of bytes from the top of the structure to this element           */
    rt++;                                  /* Short - last element      */
    size_above = (ST_CHAR *)&s.s - (ST_CHAR *)&s;
    rt->el_size = sizeof (s) - size_above;

    rt++;
    rt->el_size = 0;                   /* Structure end               */
}
```

Advanced Data Alignment — Alternate Access

The MMS-EASE AlternateAccess data conversion facility allows local data to be accessed as “packed” or “not packed” data. When data is “packed”, it means that it is present in local memory in the form represented by the derived type (original type + alternate access = derived type). When data is “not packed,” it is present in local memory specified by the original type.

Alternate Access Description Language (ADL)

The Alternate Access Description Language (**ADL**) provides an easy way to fill the abstract syntax of the Alternate Access. It will be an ASCII string that you would need to create. The **`ms_adl_to_struct`** function is provided for filling the abstract syntax of Alternate Access described by ISO spec.

Structure Control

ADL uses pre-defined sequences of characters to signal the beginning and end of structures and arrays. The following is a description of the various structure control character sequences and what they mean to ADL.

[] The brace marks are used for array access or for the nesting definitions. They signal the beginning “**[**” and the end “**]**” of a definition.

<> The angle brackets are used for named alternate access. They signal the beginning “<” and the end “>” of the name of an individual element termed as ISO identifier.

: A colon is used for the index of an alternate access.

* An asterisk is used as an index to represent all elements.

, A comma is used to separate between different alternate accesses or for the index range.

ISO identifier: This must be Visible String of no longer than 32 characters existing only of numbers (0-9), upper case letters (A-Z), lower case letters (a-z), and the **_** and **$** characters. It must not start with a number.

The **A**lternate Access **D**escription **L**anguage (ADL) provides an easy way to fill the abstract syntax of Alternate Access.

Some Examples Of Alternate Access And ADL

Below are examples of AlternateAccess and ADL. The original type indicates the beginning data structure. It is then displayed as TDL. AlternateAccess shows what portion of the data structure is to be extracted. The ADL representation is then shown along with the derived type.

Select a single named component from a structure:

Original Type

```
struct exstr
   {
   ST_INT16 comp1;
   ST_INT16 comp2;
   ST_INT16 comp3;
   };
```

TDL `{(comp1)Short, (comp2)Short, (comp3)Short}`

AlternateAccess `AA_COMP  comp = comp2`

ADL `comp2`

Derived type `Short`

Select two named components from a structure (reordered data):

Original Type
```
struct exstr
  {
  ST_INT16 comp1;
  ST_INT16 comp2;
  ST_INT16 comp3;
  };
```

TDL
```
{(comp1)Short, (comp2)Short, (comp3)Short}
```

AlternateAccess:
```
AA_COMP             comp = comp3
AA_COMP             comp = comp2
```

ADL
```
comp3,comp2
```

Derived type
```
{Short,Short}
```

Select a single element from an array:

Original Type
```
ST_INT16 arr[10];
```

TDL
```
(arr)[10:Short]
```

Alternate Access
```
       AA_INDEX     index = 3
```

ADL
```
[3]
```

Derived type
```
Short
```

Select multiple elements from an array:

Original Type
```
ST_INT16 arr[10];
```

TDL
```
(arr)[10:Short]
```

AlternateAccess
```
AA_INDEX_RANGE      low_index = 2, num_elmnts = 2
```

ADL
```
[2,2]
```

Derived type
```
[2:Short]
```

Select one component from a single element from an array of structures:

Original Type
```
struct exstr
  {
  ST_INT16 comp1;
  ST_INT16 comp2;
  ST_INT16 comp3;
  } arrstr[10];
```

TDL
```
[10:{(comp1)Short,(comp2)Short,(comp3)Short}]
```

AlternateAccess
```
       AA_INDEX_NESTindex = 3
AA_COMP             comp = comp2
AA_END_NEST
```

ADL
```
[3:comp2]
```

Derived type
```
Short
```

Select one component from multiple elements from an array of structures :

Original Type
```
struct exstr
  {
  ST_INT16 comp1;
  ST_INT16 comp2;
  ST_INT16 comp3;
  } arrstr[10];
```

TDL
```
[10:{(comp1)Short,(comp2)Short,(comp3)Short}]
```

AlternateAccess
```
AA_INDEX_RANGE_NEST low_index = 6, num_elmnts = 2
AA_COMP                    comp = comp3
AA_END_NEST
```

ADL
```
[6,2:comp2]
```

Derived type
```
[2:Short]
```

Select multiple named components from multiple elements from an array of structures:

Original Type
```
struct exstr
  {
  ST_INT16 comp1;
  ST_INT16 comp2;
  ST_INT16 comp3;
  } arrstr[10];
```

TDL
```
[10:{(comp1)Short,(comp2)Short,(comp3)Short}]
```

AlternateAccess
```
AA_INDEX_RANGE_NEST low_index = 6, num_elmnts = 2
AA_COMP                    comp = comp1
AA_COMP                    comp = comp3
AA_END_NEST
```

ADL
```
[6,2:comp1,comp3]
```

Derived type
```
[2:Short]
```

Select multiple named components from all elements from an array of structures:

Original Type
```
struct exstr
  {
  ST_INT16 comp1;
  ST_INT16 comp2;
  ST_INT16 comp3;
  } arrstr[10];
```

TDL
```
[10:{(comp1)Short,(comp2)Short,(comp3)Short}]
```

AlternateAccess
```
AA_ALL_NEST
AA_COMP             comp = comp1
AA_COMP             comp = comp3
AA_END_NEST
```

ADL
```
[*:comp1,comp3]
```

Derived type
```
[10:{Short,Short}]
```

Select single element from a single structure component that is an array :

Original Type
```
struct exstr_2
  {
  ST_INT16 arr1[23];
  ST_INT16 arr2[41];
  } strarr;
```

TDL
```
{(arr1)[23:Short],(arr2)[41:Short]}
```

Alternate Access
```
AA_COMP_NEST comp = arr1
AA_INDEX           index = 6
AA_END_NEST
```

ADL
```
arr1[6]
```

Derived Type
```
Short
```

Select multiple elements from a single structure component that is an array :

Original Type
```
struct exstr_2
  {
  ST_INT16 arr1[23];
  ST_INT16 arr2[41];
  } strarr;
```

TDL
```
{(arr1)[23:Short],(arr2)[41:Short]}
```

Alternate Access
```
AA_COMP_NEST comp = arr1
AA_INDEX_RANGE     low_index = 6, num_elmnts = 2
AA_END_NEST
```

ADL
```
arr1[6,2]
```

Derived Type
```
[Short,Short]
```

Select a single element from a two dimensional array:

Original Type
```
ST_INT16 arr2dim[5][10];
```

TDL
```
[5:[10:Short]]
```

Alternate Access
```
AA_INDEX_NESTindex = 3
AA_INDEX           index = 8
AA_END_NEST
```

ADL
```
[3[8]]
```

Derived type
```
Short
```

Select multiple elements from each dimension of a two dimensional array :

Original Type
```
ST_INT16 arr2dim[5][10];
```

TDL
```
[5:[10:Short]]
```

Alternate Access
```
AA_INDEX_RANGE_NEST low_index = 1, num_elmnts = 2
AA_INDEX_RANGE            low_index = 5, num_elmnts = 4
AA_END_NEST
```

ADL
```
[1,2[5,4]]
```

Derived type
```
[2:[4:Short]]
```

Select a single component from a structure within a structure within an array of structures:

Original Type

```
struct
  {
  ST_INT32;
  struct
    {
    struct
      {
      ST_UCHAR  ub;
      ST_UINT32 ul;
      } arr_str[10];
    }str2
  }str1;
```

TDL

```
{(l)Long,(str2){(arr_str)[10:{(ub)Ubyte,(ul)Ulong}]}}
```

Using Alternate Access, select `l & str2.arr_str[2-3].ul`

Alternate Access

```
AA_COMP               comp = l
AA_COMP_NEST   comp = str2
AA_COMP_NEST   comp = arr_str
AA_INDEX_RANGE_NEST low_index = 2, num_elmnts = 2
AA_COMP               comp = ul
AA_END_NEST
AA_END_NEST
AA_END_NEST
```

ADL

```
l,str2[arr_str[2,2:ul]]
```

Derived Type

```
{Long,[2:Long]}
```

Original Type

```
struct
  {
  ST_INT32 l;
  struct
    {
    ST_UINT8  ub1;
    ST_UINT32 ul;
    } arr_str[10];
  ST_UINT8     ub2;
  }str3;
```

TDL

```
{(l)Long,(arr_str)[10:{(ub1)Ubyte,(ul)Ulong}],(ub2)Ubyte}
```

Using Alternate Access, select `ub2 & arr_str[2-3].ul`

Alternate Access

```
AA_COMP               comp = ub2
AA_COMP_NEST   comp = arr_str
AA_INDEX_RANGE_NEST low_index = 2, num_elmnts = 2
AA_COMP               comp = ul
AA_END_NEST
AA_END_NEST
```

ADL

```
ub2,arr_str[2,2:ul]
```

Derived Type

```
{Ubyte,[2:Ulong]}
```

Original Type

```
struct
  {
  struct
    {
    struct
      {
      ST_UINT8 ub1;
      ST_UINT8 ul1;
      }arr_str11[5];
    ST_UINT32 ub11;
    }arr_str1[5];
    struct
      {
      ST_UINT8  ub2;
      ST_UTIN32 ul2;
      } arr_str2[5];
  };
```

TDL

```
{(arr_str1)[5:{(arr_str11)[5:{(ub1)Ubyte,(ul1)Ulong}],
(ub11)Ulong}],(arr_str2)[5:{(ub2)Ubyte,(ul2)Ulong}]}
```

Using Alternate Access, select `arr_str1[ALL].arr_str11[3].ul1 & arr_str2[2-2].ul2`

Alternate Access

```
AA_COMP_NEST        comp = arr_str1
AA_ALL_NEST
AA_COMP_NEST        comp = arr_str11
AA_INDEX_NEST       index = 3
AA_COMP                    comp = ul1
AA_END_NEST
AA_END_NEST
AA_END_NEST
AA_END_NEST
AA_COMP_NEST        comp = arr_str2
AA_INDEX_RANGE_NEST        low_index = 2, num_elmnts = 2
AA_COMP                    comp = ul2
AA_END_NEST
AA_END_NEST
```

ADL

```
arr_str1[*:arr_str11[3:ul1]],arr_str2[2,2:ul2]
```

Derived Type

```
{[5:Ulong],[2:Ulong]}
```

Original Type

```
struct
  {
  ST_INT32 l;
  struct
    {
    ST_INT8   ub1;
    ST_UINT32 ul1;
    }comp1[5];
  struct
    {
    ST_UINT8  ub2;
    ST_UINT32 ul2;
    }comp2[5];
  }arr_str[5];
```

TDL

```
[5:{(l)Long,(comp1)[5:{(ub1)Ubyte,(ul1)Ulong}],
      (comp2)[5: {(ub2)Ubyte,(ul2)Ulong}]}]
```

Using Alternate Access, select `[ALL].comp1[2].ul1  & [ALL].comp2[2-3].ul2`

Alternate Access

```
AA_ALL_NEST
AA_COMP_NEST          comp = comp1
AA_INDEX_NEST         index = 2
AA_COMP                     comp = ul1
AA_END_NEST
AA_END_NEST
AA_COMP_NEST          comp = comp2
AA_INDEX_RANGE_NEST         low_index = 2, num_elmnts = 2
AA_COMP                     comp = ul2
AA_END_NEST
AA_END_NEST
AA_END_NEST
```

ADL

```
[*:comp1[2:ul1],comp2[2,2:ul2]]
```

Derived Type

```
[5:{Ulong,[2:ulong]}]
```

Data Manipulation

Certain support functions are also supplied with MMS-EASE to perform conversions between ASN.1, the local representation of a variable's data, and its runtime type. These functions are described on the following pages.

The **NAMED_TYPE** structure is used to determine the type of data to be converted. These functions depend on the Virtual Machine Database, and the functions that manage that database. The data manipulation functions are also used internally by other virtual machine functions, such as **mv_read_resp**. They are exposed to the user mainly to augment the PPI. Applications that use the main VMI functions for variable access would not need to use these support functions.

As mentioned on page 2-81, MMS-EASE assumes no knowledge of the native C compiler's structure layout algorithm. It will not account for any pad bytes that may be transparently added by the compiler.

Process Arbitrary Data Structure

The following structure is used in the **ms_process_arb_data** function.

```
typedef struct m_arb_data_ctrl
  {
  ST_RET (*arrStart) (RT_AA_CTRL *rtaa);
  ST_RET (*arrEnd)   (RT_AA_CTRL *rtaa);
  ST_RET (*strStart) (RT_AA_CTRL *rtaa);
  ST_RET (*strEnd)   (RT_AA_CTRL *rtaa);
  ST_RET (*int8)     (ST_INT8    *data_dest, RT_AA_CTRL *rtaa);
  ST_RET (*int16)    (ST_INT16   *data_dest, RT_AA_CTRL *rtaa);
  ST_RET (*int32)    (ST_INT32   *data_dest, RT_AA_CTRL *rtaa);
  ST_RET (*int64)    (ST_INT64   *data_dest, RT_AA_CTRL *rtaa);
  ST_RET (*uint8)    (ST_UINT8   *data_dest, RT_AA_CTRL *rtaa);
  ST_RET (*uint16)   (ST_UINT16  *data_dest, RT_AA_CTRL *rtaa);
  ST_RET (*uint32)   (ST_UINT32  *data_dest, RT_AA_CTRL *rtaa);
  ST_RET (*uint64)   (ST_UINT64  *data_dest, RT_AA_CTRL *rtaa);
  ST_RET (*flt)      (ST_FLOAT   *data_dest, RT_AA_CTRL *rtaa);
  ST_RET (*dbl)      (ST_DOUBLE  *data_dest, RT_AA_CTRL *rtaa);
  ST_RET (*oct)      (ST_UCHAR   *data_dest, RT_AA_CTRL *rtaa);
  ST_RET (*bool)     (ST_BOOLEAN *data_dest, RT_AA_CTRL *rtaa);
  ST_RET (*bcd1)     (ST_INT8    *data_dest, RT_AA_CTRL *rtaa);
  ST_RET (*bcd2)     (ST_INT16   *data_dest, RT_AA_CTRL *rtaa);
  ST_RET (*bcd4)     (ST_INT32   *data_dest, RT_AA_CTRL *rtaa);
  ST_RET (*bs)       (ST_UCHAR   *data_dest, RT_AA_CTRL *rtaa);
  ST_RET (*vis)      (ST_CHAR    *data_dest, RT_AA_CTRL *rtaa);
  ST_RET (*bt4)      (ST_INT32   *data_dest, RT_AA_CTRL *rtaa);
  ST_RET (*bt6)      (ST_INT32   *data_dest, RT_AA_CTRL *rtaa);
  ST_RET (*gt)       (time_t     *data_dest, RT_AA_CTRL *rtaa);
  } M_ARB_DATA_CTRL;
```

If the application returns a value other than **SD_SUCCESS** from any one of the following functions, **ms_process_arb_data** will stop accessing the data and return **SD_FAILURE** to the application.

Fields:

`arrStart` This function is called when the start of a MMS array is encountered in the derived MMS type. A pointer to the **RT_AA_CTRL** structure representing the start of the MMS array is supplied as an argument.

`arrEnd` This function is called when the end of a MMS array is encountered in the derived MMS type. A pointer to the **RT_AA_CTRL** structure representing the end of the MMS array is supplied as an argument.

`strStart` This function is called when the start of a MMS structure is encountered in the derived MMS type. A pointer to the **RT_AA_CTRL** structure representing the start of the MMS structure is supplied as an argument.

`strEnd` This function is called when the start of a MMS structure is encountered in the derived MMS type. A pointer to the **RT_AA_CTRL** structure representing the end of the MMS structure is supplied as an argument.

`int8` This function is called when an 8 bit signed integer is encountered in the derived MMS type. A pointer to the beginning of the 8 bit integer and a pointer to the **RT_AA_CTRL** structure representing the 8 bit integer in the MMS Data are supplied as arguments.

`int16` This function is called when a 16 bit signed integer is encountered in the derived MMS type. A pointer to the beginning of the 16 bit integer and a pointer to the **RT_AA_CTRL** structure representing the 16 bit integer in the MMS Data are supplied as arguments.

`int32` This function is called when a 32 bit signed integer is encountered in the derived MMS type. A pointer to the beginning of the 32 bit integer and a pointer to the **RT_AA_CTRL** structure representing the 32 bit integer in the MMS Data are supplied as arguments.

`int64` This function is called when a 64 bit signed integer is encountered in the derived MMS type. A pointer to the beginning of the 64 bit integer and a pointer to the **RT_AA_CTRL** structure representing the 64 bit integer in the MMS Data are supplied as arguments.

`uint8` This function is called when an 8 bit unsigned integer is encountered in the derived MMS type. A pointer to the beginning of the 8 bit integer and a pointer to the **RT_AA_CTRL** structure representing the 8 bit unsigned integer in the MMS Data are supplied as arguments.

`uint16` This function is called when a 16 bit unsigned integer is encountered in the derived MMS type. A pointer to the beginning of the 16 bit integer and a pointer to the **RT_AA_CTRL** structure representing the 16 bit unsigned integer in the MMS Data are supplied as arguments.

`uint32` This function is called when a 32 bit unsigned integer is encountered in the derived MMS type. A pointer to the beginning of the 32 bit integer and a pointer to the **RT_AA_CTRL** structure representing the 32 bit unsigned integer in the MMS Data are supplied as arguments.

`uint64` This function is called when a 64 bit unsigned integer is encountered in the derived MMS type. A pointer to the beginning of the 64 bit integer and a pointer to the **RT_AA_CTRL** structure representing the 64 bit unsigned integer in the MMS Data are supplied as arguments.

`flt` This function is called when a single precision floating point element is encountered in the derived MMS type. A pointer to the beginning of the float and a pointer to the **RT_AA_CTRL** structure representing the single precision float in the MMS Data are supplied as arguments.

`dbl` This function is called when a double precision floating point element is encountered in the derived MMS type. A pointer to the beginning of the double and a pointer to the **RT_AA_CTRL** structure representing the double precision float in the MMS Data are supplied as arguments.

`oct` This function is called when an octet string element is encountered in the derived MMS type. A pointer to the beginning of the octet string and a pointer to the **RT_AA_CTRL** structure representing the octet string in the MMS Data are supplied as arguments. The **RT_AA_CTRL** information can be used to see if the octet string is a positive fixed or a negative variable length.

`bool` This function is called when a single byte boolean element is encountered in the derived MMS type. A pointer to the beginning of the boolean and a pointer to the **RT_AA_CTRL** structure representing the boolean in the MMS Data are supplied as arguments.

`bcd1` This function is called when a single byte binary coded data element is encountered in the derived MMS type. A pointer to the beginning of an 8 bit integer and a pointer to the **RT_AA_CTRL** structure representing the single byte BCD in the MMS Data are supplied as arguments. A BCD1 value in this function, is a single byte integer which may only contain values in the range [0..99].

`bcd2` This function is called when a double byte binary coded data element is encountered in the derived MMS type. A pointer to the beginning of the 16 bit integer and a pointer to the **RT_AA_CTRL** structure representing the double byte BCD in the MMS Data are supplied as arguments. A BCD2 value in this function, is a two byte integer which may only contain values in the range [0.9999].

`bcd4` This function is called when a four byte binary coded data element is encountered in the derived MMS type. A pointer to the beginning of a 32 bit integer and a pointer to the **RT_AA_CTRL** structure representing the double byte BCD in the MMS Data are supplied as arguments. A BCD4 value in this function, is a four byte bit integer which may only contain values in the range [0..99999999].

`bs` This function is called when a bit string element is encountered in the derived MMS type. A pointer to the beginning of the bit string and a pointer to the **RT_AA_CTRL** structure representing the bit string in the MMS Data are supplied as arguments. The **RT_AA_CTRL** information can be used to see if the bit string is a positive fixed or a negative variable length.

`vis` This function is called when a visible string element is encountered in the derived MMS type. A pointer to the beginning of the visible string and a pointer to the **`RT_AA_CTRL`** structure representing the visible string in the MMS Data are supplied as arguments. The **`RT_AA_CTRL`** information can be used to see if the visible string is a positive fixed or a negative variable length.

`bt4` This function is called when a four octet Binary Time Of Day element is encountered in the derived MMS type. A pointer to the beginning of a **`ST_INT32`** and a pointer to the **`RT_AA_CTRL`** structure representing the Binary Time Of Day in the MMS Data are supplied as arguments.

`bt6` This function is called when a six octet Binary Time Of Day element is encountered in the derived MMS type. A pointer to the beginning of an array of **`ST_INT32`**s and a pointer to the **`RT_AA_CTRL`** structure representing the Binary Time Of Day in the MMS Data are supplied as arguments. The first **`ST_INT32`** in the array represents milliseconds since midnight of the current day. The second **`ST_INT32`** in the array represents days since January 1 1984.

`gt` This function is called when a Generalized Time element is encountered in the derived MMS type. A pointer to the beginning of a **`time_t`** structure and a pointer to the **`RT_AA_CTRL`** structure representing the generalized time in the MMS Data are supplied as arguments.

Runtime Alternate Access Data Structure

This is the structure used in table form to implement alternate access from the Server perspective. It allows reordering of the elements as well as limited selection.

```
struct rt_aa_ctrl
  {
  RUNTIME_TYPE *rt;
  ST_UINT      offset_to_data;
  ST_INT       el_size;
  };
typedef struct rt_aa_ctrl RT_AA_CTRL;
```

Fields:

`rt` This pointer to a structure of type **`RUNTIME_TYPE`** contains the runtime type definition. See page 2-99 for a definition of this structure.

`offset_to_data` This indicates the offset to data element.

`el_size` This indicates the size of the element in memory.

Data Manipulation Functions

ms_aa_to_adl

Usage: This function extracts information from an alternate access structure and makes an ADL string as output. It assumes that the buffer **adl_str** has been allocated before calling this function. This buffer will be filled up to the value indicated by **max_adl_len**.

Function Prototype:

```
ST_RET ms_aa_to_adl (ALT_ACCESS *alt_acc,
                     ST_CHAR *adl_str,
                     ST_INT max_adl_len);
```

Parameters:

alt_acc	This pointer to the **ALT_ACCESS** structure contains the alternate access information corresponding to the data to be converted.
adl_str	This is a pointer to the ADL string.
max_adl_len	This contains the maximum length of the ADL string.

Return Value:

ST_RET	**SD_SUCCESS**.	No Error. Data converted.
	<> 0	Error. Data not converted.

ms_aa_to_asn1

Usage: This support function is provided to create an ASN.1 encoded Alternate Access corresponding to an **ALT_ACCESS** structure. The ASN.1 function **strt_asn1_bld** must be called to initialized the encoding buffer before calling this function. The global variable **asn1_field_ptr** is used to find the beginning of the encoding.

Function Prototype:

```
ST_RET ms_aa_to_asn1 (ALT_ACCESS *alt_acc);
```

Parameters:

`alt_acc` This pointer to an **ALT_ACCESS** structure contains the alternate access information to be converted to an ASN.1 form.

Return Value: `ST_RET`

SD_SUCCESS. No Error. Data converted.

`<> 0` Error. Data not converted.

Example:

```
strt_asn1_bld (aa_buf, aa_buf_size);
  if (ms_aa_to_asn1 (&vmi->i.alt_acc) = SD_SUCCESS)
  {
  asn1_start = asn1_field_ptr+1;
  asn1_len = (aa_buf + aa_buf_size) - asn1_start;
  vl->alt_access.len = asn1_len;
  vl->alt_access.data = asn1_start;
  vl->alt_access_pres = SD_TRUE;
  }
```

ms_adl_to_aa

Usage: This function parses an ADL (AlternateAccess Definition Language) string and produces a corresponding alternate access structure of type, **ALT_ACCESS**. This function calls **chk_malloc** to allocate an array of **ALT_ACC_EL** structures. It assumes that the buffer used to create this structure will be freed by the application after its usage.

Function Prototype:

```
ST_RET ms_adl_to_aa (ST_CHAR *adl_str,
                     ALT_ACCESS *alt_acc);
```

Parameters:

adl_str	This is a pointer to the ADL string to be converted.
alt_acc	This pointer to the **ALT_ACCESS** structure contains the alternate access information of the converted data.

Return Value: ST_RET

SD_SUCCESS. No Error. Data converted.

SD_FAILURE. Error. Data not converted.

ms_asn1_data_to_locl

Usage: This support function can be called to convert an ASN.1encoded variable data to local format in an area pointed to by the **data_dest** argument. The MMS type may be supplied to this function as either known or unknown. **RUNTIME_TYPE** information supplied in the **rt** and **t_len** arguments are used in the conversion if they are available. When the MMS Type information is not known, this function can be used to automatically derive the MMS Type from the ASN.1 encoded data and allocate the space required to convert the encoding into local format. Both the **RUNTIME_TYPE** and **data_dest** are returned to the application. See page 2-89 for more information on the ASN.1 data representation specified by MMS.

Function Prototype:

```
ST_RET ms_asn1_data_to_locl (ST_UCHAR *asn1_data,
                             ST_INT asn1_data_len,
                             ST_VOID **data_dest,
                             ST_INT *data_dest_len,
                             RUNTIME_TYPE **rt,
                             ST_INT *t_len);
```

Parameters:

`asn1_data` This is a pointer to the ASN.1 data representation.

`asn1_data_len` This is the length in bytes of the data pointed to by **asn1_data**.

`data_dest` This is a reference to a pointer that may be supplied to reference a location into where the application would like the ASN.1 data decoded. When this argument is supplied as null, the function will allocate the space required to decode the data and return the pointer as an output parameter.

`data_dest_len` This is the length in bytes of the data pointed to by **data_dest**.

`rt` This pointer to an array of **RUNTIME_TYPE** structures contains the runtime type information of the ASN.1 encoded data. If the argument is supplied as a null, the assumption is that the application does not know what type of data is contained in the encoding. The function will allocate the space for the derived MMS type and return the pointer as an output parameter. The application may examine this output information to determine what type of data was decoded.

`t_len` This points to the number of elements in the **RUNTIME_TYPE** array.

Return Value: `ST_RET`

SD_SUCCESS. Data converted.

SD_FAILURE. Data not converted.

ms_asn1_data_to_runtime

Usage: This support function can be called to derive a MMS Type representation from an ASN.1 encoding of variable data. An array of **RUNTIME_TYPE** elements is returned to the application MMS variable data is not converted to local format. The benefit of this support function lies in being able to derive the MMS type from an arbitrary encoding of MMS Data. See page 2-89 for more information on the ASN.1 data representation specified by MMS.

Function Prototype:

```
ST_RET ms_asn1_data_to_runtime (RUNTIME_TYPE **tptr,
                                ST_INT *t_len,
                                ST_UCHAR *asn1ptr,
                                ST_INT asn1_len);
```

Parameters:

`tptr` This is the address of a pointer that will be filled in by the function to reference an array of **RUNTIME_TYPE** structures. These structures contain the runtime type information associated with the encoded MMS Data and consist of dynamic memory drafted by MMS-EASE. Failure to release the memory by calling **chk_free** will result in a memory leak.

`t_len` This is the address of a **ST_INT** that will be filled in by the function with the number of **RUNTIME_TYPE** structures in the **tptr** array.

`asn1ptr` This is a pointer to the ASN.1 data representation.

`asn1_len` This is the length in bytes of the data pointed to by **asn1ptr**.

Return Value: `ST_RET`

SD_SUCCESS. Data converted.

SD_FAILURE. Data not converted.

ms_asn1_to_aa

Usage: This support function can be called to convert an ASN.-encoded AlternateAccess to the run-time AlternateAccess representation. This function uses a pointer to an **ALT_ACCESS** structure as a place to store the converted AlternateAccess. The array of **ALT_ACC_EL** structures allocated by this function should be released by the application by calling **chk_free** at a later time. See page 2-89 for more information on the ASN.1 data representation specified by MMS.

Function Prototype:

```
ST_RET ms_asn1_to_aa (ST_UCHAR *asn1ptr,
                      ST_INT asn1len,
                      ALT_ACCESS *alt_acc_out);
```

Parameters:

`asn1ptr` This is a pointer to the ASN.1 encoded Alternate Access representation. This could be the same pointer as **alternate_access.data**.

`asn1len` This is the length of bytes of the data pointed to by **asn1ptr**. This could be the same as **alternate_access.len**.

`alt_acc_out` This pointer to the **ALT_ACCESS** structure contains the alternate access information corresponding to the data to be converted.

Return Value: `ST_RET`

SD_SUCCESS. No Error. Data converted.

SD_FAILURE. Error. Data not converted.

ms_asn1_to_local

Usage: This support function can be called to convert an ASN.1 Representation of a variable's data into the local representation with which the variable actually exists in memory using the runtime type. This function uses a pointer to a **RUNTIME_TYPE** structure to determine how to convert the data from the ASN.1 to the local representation. If using the virtual machine for variable operations, this function is not needed.

Function Prototype:

```
ST_RET ms_asn1_to_local (RUNTIME_TYPE *rt,
                         ST_INT num_rt,
                         ST_UCHAR *asn1,
                         ST_INT asn1_len,
                         ST_CHAR *dest);
```

Parameters:

`rt` This pointer to the **RUNTIME_TYPE** structure contains the runtime type information corresponding to the data to be converted.

`num_rt` This indicates the number of runtime types to be converted.

`asn1` This is a pointer to the ASN.1 data representation. This could be the same pointer as **var_acc_data.data**.

`asn1_len` This is the length of bytes of the data pointed to by ***asn1**. This could be the same as **var_acc_data.len**.

`dest` This is a pointer to the area in memory in which the local representation is to be put.

Return Value: `ST_RET` **SD_SUCCESS**. No Error. Data converted.

SD_FAILURE. Error. Data not converted.

ms_asn1_to_local_aa

Usage: This support function is the same as **ms_asn1_to_locl_aa** except that the type information is supplied directly by a runtime type. It can be called to convert an ASN.1 Representation of a variable's data into the local representation with which the variable actually exists in memory using the runtime type. This function uses a pointer to a **RUNTIME_TYPE** structure to determine how to convert the data from the ASN.1 to the local representation. If using the virtual machine for variable operations, this function is not needed.

Function Prototype:

```
ST_RET ms_asn1_to_local_aa (RUNTIME_TYPE *rt_head,
                            ST_INT rt_num,
                            ALT_ACCESS *alt_acc,
                            ST_UCHAR *asn1ptr,
                            ST_INT asn1len,
                            ST_CHAR *dptr);
```

Parameters:

`rt_head` This pointer to the **RUNTIME_TYPE** structure contains the beginning of the runtime type information corresponding to the data to be converted.

`rt_num` This indicates the number of runtime types to be converted.

`alt_acc` This pointer to an **ALT_ACCESS** structure contains the alternate access information corresponding to the data to be converted.

`asn1ptr` This is a pointer to the ASN.1 data representation. This could be the same pointer as **var_acc_data.data**.

`asn1len` This is the length of bytes of the data pointed to by **asn1**. This could be the same as **var_acc_data.len**.

`dptr` This is a pointer to the area in memory in which the local representation is to be put.

Return Value: `ST_RET`

SD_SUCCESS. No Error. Data converted.

SD_FAILURE. Error. Data not converted.

ms_asn1_to_locl

Usage: This support function can be called to convert an ASN.1 Representation of a variable's data into the local representation with which the variable actually exists in memory. This function uses a pointer to a **NAMED_TYPE** structure to determine how to convert the data from the ASN.1 to the local representation. If using the virtual machine for variable operations, this function is not needed. However, if there is a variable's data stored in ASN.1 in some other structure, such as a journal entry, you need to convert from ASN.1 into the local representation. See page 2-89 for more information on the ASN.1 data representation specified by MMS.

Function Prototype:

```
ST_RET ms_asn1_to_locl (NAMED_TYPE *type,
                        ST_UCHAR *asn1,
                        ST_INT asn1_len,
                        ST_CHAR *dest);
```

Parameters:

type	This pointer to the **NAMED_TYPE** structure contains the type information corresponding to the data to be converted.
asn1	This is a pointer to the ASN.1 data representation. This could be the same pointer as **var_acc_data.data**.
asn1len	This is the length of bytes of the data pointed to by **asn1**. This could be the same as **var_acc_data.len**.
dest	This is a pointer to the area in memory in which the local representation is to be put.

Return Value:

ST_RET	**SD_SUCCESS**. No Error. Data converted.
	SD_FAILURE. Error. Data not converted.

ms_asn1_to_locl_aa

Usage: This support function can be called to convert an ASN.1 Representation of a variable's data when a Named Type for the data element and an Alternate Access is present into the local representation with which the variable actually exists in memory. This function uses a pointer to a **NAMED_TYPE** structure to determine how to convert the data from the ASN.1 to the local representation. If using the virtual machine for variable operations, this function is not needed. However, if there is a variable's data stored in ASN.1 in some other structure, such as a journal entry, you need to convert from ASN.1 into the local representation. See page 2-89 for more information on the ASN.1 data representation specified by MMS.

Function Prototype:

```
ST_RET ms_asn1_to_locl_aa (NAMED_TYPE *tptr,
                           ALT_ACCESS *alt_acc,
                           ST_UCHAR *asn1ptr,
                           ST_INT asn1len,
                           ST_CHAR *dptr);
```

Parameters:

tptr	This pointer to a **NAMED_TYPE** structure contains the type information corresponding to the data to be converted.
alt_acc	This pointer to an **ALT_ACCESS** structure contains the alternate access information corresponding to the data to be converted.
asn1ptr	This is a pointer to the ASN.1 data representation. This could be the same pointer as **var_acc_data.data**.
asn1len	This is the length of bytes of the data pointed to by ***asn1**. This could be the same as **var_acc_data.len**.
dptr	This is a pointer to the area in memory in which the local representation is to be put.

Return Value: ST_RET

SD_SUCCESS. No Error. Data converted.

SD_FAILURE. Error. Data not converted.

ms_asn1_to_runtime


Usage: This support function can be called to convert an ASN.1 Representation of a variable's data into the runtime type. This function uses a pointer to a **RUNTIME_TYPE** structure to determine how to convert the data from the ASN.1 to the runtime type. ~~See page 2-89~~ for more information on the ASN.1 data representation specified by MMS.

Function Prototype:

```
ST_INT ms_asn1_to_runtime (ST_UCHAR *asn1,
                           ST_UINT asn1len,
                           RUNTIME_TYPE *dest,
                           ST_INT num_rt_dest);
```

Parameters:

asn1	This is a pointer to the ASN.1 data representation. This could be the same pointer as **var_acc_data.data**.
asn1len	This is the length of bytes of the data pointed to by **asn1**. This could be the same as **var_acc_data.len**.
dest	This pointer to a **RUNTIME_TYPE** structure contains the runtime type information corresponding to the data to be converted.
num_rt_dest	This is the number of runtime types present in **dest**.

Return Value: ST_INT

SD_SUCCESS. No Error. Data converted.

SD_FAILURE. Error. Data not converted.

ms_asn1_to_tdl

Usage: This support function can be called to convert an ASN.1 Representation of a variable's data into SISCO's Type Definition Language (TDL). This function uses a pointer to a **TIME_TYPE** structure to determine how to convert the data from the ASN.1 to the runtime type. See page 2-89 for more information on the ASN.1 data representation specified by MMS.

Function Prototype:

```
ST_CHAR *ms_asn1_to_tdl (ST_UCHAR *asn1_ptr,
                         ST_INT asn1_len,
                         ST_INT max_tdl_len);
```

Parameters:

asn1_ptr	This is a pointer to the ASN.1 data representation. This could be the same pointer as **var_acc_data.data**.
asn1_len	This is the length of bytes of the data pointed to by **asn1_ptr**. This could be the same as **var_acc_data.len**.
max_tdl_len	This indicates the maximum length of the TDL to be converted.

Return Value:

ST_CHAR *	This is a pointer to the beginning of the converted TDL. In case of an error, the pointer is set to null and **mms_op_err** is written with the error code.

ms_local_to_asn1

Usage: This support function can be called to convert a local variable's data from runtime type into the ASN.1-encoded format of that data. It uses a pointer to a **RUNTIME_TYPE** structure to determine how to convert the data from the variable's local runtime type representation into the ASN.1-encoded format. If using the virtual machine for variable operations, this function is not needed. However, if a variable's data is to be stored in some other structure, such as a journal entry, the local representation will need to be converted into the ASN.1 representation.

Function Prototype:

```
ST_RET ms_local_to_asn1 (RUNTIME_TYPE *rt,
                         ST_INT num_rt,
                         ST_CHAR *src);
```

Parameters:

`rt` This pointer to a **RUNTIME_TYPE** structure contains the runtime type information corresponding to the data to be converted.

`num_rt` This indicates the number of runtime types present in this data.

`src` This is a pointer to the area in memory in which the local representation exists.

Return Value: `ST_RET`

SD_SUCCESS. No Error. Data converted.

SD_FAILURE. Error. Data not converted.

ms_local_to_asn1_aa

Usage: This function is the same as **ms_locl_to_asn1_aa** except that the type information is supplied directly by a runtime type. It uses a pointer to a **RUNTIME_TYPE** structure to determine how to convert the data from the variable's local runtime type representation into the ASN.1-encoded format. If using the virtual machine for variable operations, this function is not needed. However, if a variable's data is to be stored in some other structure, such as a journal entry, the local representation will need to be converted into the ASN.1 representation.

Function Prototype:

```
ST_RET ms_local_to_asn1_aa (RUNTIME_TYPE *rt_head,
                            ST_INT rt_num,
                            ALT_ACCESS *alt_acc,
                            ST_CHAR *dptr);
```

Parameters:

rt_head	This pointer to a **RUNTIME_TYPE** structure contains the beginning of the runtime type information corresponding to the data to be converted.
rt_num_rt	This indicates the number of runtime types present in this data.
aa	This pointer to an **ALT_ACCESS** structure contains the alternate access information corresponding to the data to be converted
dptr	This is a pointer to the area in memory in which the local representation will exist.

Return Value: ST_RET

SD_SUCCESS. No Error. Data converted.

SD_FAILURE. Error. Data not converted.

ms_locl_to_asn1

Usage: This support function can be called to convert a local variable's data into the ASN.1-encoded format of that data. It uses a pointer to a **NAMED_TYPE** structure to determine how to convert the data from the variable's local representation into the ASN.1-encoded format. If using the virtual machine for variable operations, this function is not needed. However, if a variable's data is to be stored in some other structure, such as a journal entry, the local representation will need to be converted into the ASN.1 representation.

Function Prototypes: `ST_RET ms_locl_to_asn1 (NAMED_TYPE *type, ST_CHAR *src);`

Parameters:

`type` This pointer to a **NAMED_TYPE** structure contains the type information corresponding to the data to be converted.

`src` This is a pointer to the area in memory in which the local representation exists.

Return Value: `ST_RET` **SD_SUCCESS**. No Error. Data converted.

SD_FAILURE. Error. Data not converted.

NOTE: This function uses the ASN1DE tools to perform its action. The build tools must first be initialized. Additionally, the location and length of the ASN.1 representation is passed to the application program using the ASN1DE global variable **field_ptr**. The following is a simplified example of how to use this function:

```
#define BUFFER_LEN 100
ST_CHAR asn1_buffer[BUFFER_LEN];        /* buffer to put ASN.1 in */
struct object_name type_name;    /* the name of the type    */

type = ms_find_named_type_obj(type_name,0);
                                 /* get the named type pointer */
strt_asn1_bld(asn1_buffer,BUFFER_LEN);
                                 /* initialize the ASN.1 tools */
ms_locl_to_asn1(typeptr,src);    /* convert the data        */

asn1len = (asn1_buffer + BUFFER_LEN) - field_ptr - 1;
                                 /* length of ASN.1                */
asn1ptr = field_ptr+1;                  /* pointer to ASN.1               */
```

ms_locl_to_asn1_aa

Usage: This support function is used to encode local data into ASN.1 data element when a named type for the data element and an Alternate Access are present. It uses a pointer to a **NAMED_TYPE** structure to determine how to convert the data from the variable's local representation into the ASN.1-encoded format. If using the virtual machine for variable operations, this function is not needed. However, if a variable's data is to be stored in some other structure, such as a journal entry, the local representation will need to be converted into the ASN.1 representation.

Function Prototype:

```
ST_RET ms_locl_to_asn1_aa (NAMED_TYPE *tptr,
                           ALT_ACCESS *alt_acc,
                           ST_CHAR *dptr);
```

Parameters:

`tptr`	This pointer to a **NAMED_TYPE** structure contains the type information corresponding to the data to be converted.
`alt_acc`	This pointer to an **ALT_ACCESS** structure contains the alternate access information corresponding to the data to be converted.
`dptr`	This is a pointer to the area in memory in which the local representation exists.

Return Value: `ST_RET`

SD_SUCCESS. No Error. Data converted.

SD_FAILURE. Error. Data not converted.

NOTE: This function uses the ASN1DE tools to perform its action. The build tools must first be initialized. Additionally, the location and length of the ASN.1 representation is passed to the application program using the ASN1DE global variable **field_ptr**. The following is a simplified example of how to use this function:

```
#define BUFFER_LEN 100
ST_CHAR asn1_buffer[BUFFER_LEN];        /* buffer to put ASN.1 in */
struct object_name type_name;    /* the name of the type    */

type = ms_find_named_type_obj(type_name,0);
                                 /* get the named type pointer */
strt_asn1_bld(asn1_buffer,BUFFER_LEN);
                                 /* initialize the ASN.1 tools */
ms_locl_to_asn1(typeptr,src);    /* convert the data        */

asn1len = (asn1_buffer + BUFFER_LEN) - field_ptr - 1;
                                 /* length of ASN.1                */
asn1ptr = field_ptr+1;                  /* pointer to ASN.1                */
```

ms_process_arb_data

Usage: This support function is used to process MMS data of arbitrary type for either assigning values to the data members or retrieving values from them. MMS data of complex type normally requires custom programming to assign or retrieve values from specific types of data. Using this function allows a common set of callback functions to be supplied and invoked for each purpose. Although this function can allow complete access to any MMS data type, alternate access may also be supplied, in which case only specific array elements and structure members appear in the call back functions.

Function Prototype:

```
ST_RET ms_process_arb_data (ST_CHAR *data_base,
                            RUNTIME_TYPE *rt_head,
                            ST_INT rt_num,        ST_BOO-
                            LEAN alt_acc_pres,    ST_BOO-
                            LEAN alt_acc_packed,
                            ALT_ACCESS *alt_acc,
                            M_ARB_DATA_CTRL *ac);
```

Parameters:

`data_base` This pointer references the beginning of the arbitrary data. Alternate Access information is applied to the runtime type beginning at this location in memory.

`rt_head` This pointer to a structure of type **`RUNTIME_TYPE`** contains the beginning of the runtime type definition. See page 2-99 for a definition of this structure.

`rt_num` This indicates the number of **`RUNTIME_TYPE`** elements found in the runtime type array.

`alt_acc_pres` **`SD_TRUE`**. **`alt_acc`** is present in the function call.

`SD_FALSE`. **`alt_acc`** is not present and the **`alt_acc_packed`** and **`alt_acc`** arguments are ignored.

`alt_acc_packed` **`SD_TRUE`**. The data representing a derived MMS type is packed in memory.

`SD_FALSE`. The data representing a derived MMS type is placed in memory according to its offset from the native runtime type.

`alt_acc` This structure of type **`ALT_ACCESS`** contains the alternate access description. See page 2-0 for more information on this structure.

`ac` This pointer of type **`M_ARB_DATA_CTRL`** is an array of function pointers to application specific functions that are called in the context of processing data of a specific MMS primitive type. There is a function pointer in this array for each primitive type of MMS Data. See page 2-110 for more information on this structure.

Return Value: `ST_RET` **`SD_SUCCESS`**. Data was processed successfully.

`SD_FAILURE`. Error. Data was not processed successfully.

ms_rt_size_calc

Usage: This support function applies the information found in the **m_data_algn_table** to fixup a **RUNTIME_TYPE** array and provide the offset and index information needed to access data according to the rules of the native machine, compiler, and operating system.

Function Prototype:

```
ST_VOID ms_rt_size_calc (RUNTIME_TYPE *rt_head,
                         ST_INT rt_num);
```

Parameters:

`rt_head` This pointer to the beginning of a **RUNTIME_TYPE** array.

`rt_num` This indicates the number of runtime types present in the runtime type array.

Return Value: `ST_VOID` (ignored)

ms_runtime_to_tdl

Usage: This support function can be called to convert a runtime type into SISCO's Type Definition Language (TDL). This function uses a pointer to a **RUNTIME_TYPE** structure to determine how to convert the data from the runtime type to TDL. See page 2-91 for more information on TDL.

Function Prototype:

```
ST_INT ms_runtime_to_tdl (RUNTIME_TYPE *rt,
                          ST_INT rt_num,
                          ST_CHAR *tdl_buf,
                          ST_INT max_tdl_len);
```

Parameters:

`rt` This pointer to a **RUNTIME_TYPE** structure contains the runtime type information corresponding to the data to be converted.

`num_rt` This indicates the number of runtime types present in this data.

`tdl_buf` This is a pointer to the buffer in which the place the TDL string converted from the runtime type.

`max_tdl_len` This indicates the maximum length of the TDL to be converted.

Return Value:

`ST_INT` Returns the number of characters in the TDL string.

SD_FAILURE. Error. Data not converted.

Address Resolution Functions

u_get_named_addr

Usage: This pre-named, user-defined, address resolution function is called by the MMS-EASE virtual machine when a named variable read or write is being handled by the virtual machine. It is used to resolve a named variable address. It allows the user to determine how to resolve addresses of network variables, which may be artificial. The function is passed a pointer to the structure containing the named variable definition information. You MUST write your own **u_get_named_addr** function to return a pointer to the local variable where the read or write is to occur. The length of this data is determined by the named type definition specified for that variable. After returning from this function, either the **read_ind_fun** or **write_ind_fun** function will be called, with the information provided upon returning from this function.

Function Prototype:

```
ST_CHAR *u_get_named_addr (NAMED_VAR *var);
```

Parameters:

var	This is a pointer to the **NAMED_VAR** structure for the variable being accessed.

Return Value:

ST_CHAR *	This is a character pointer to the data for the variable whose address was just resolved. The data for the local variable should exist at the location specified by this pointer. In case of an error, the pointer is set to null and **mms_op_err** is written with the error code.

NOTE: In order for the virtual machine to operate properly, this function MUST be supplied by you. In most cases, the **addr** member of the **NAMED_VAR** structure can be used to hold the address information. Simply return the contents of the appropriate member of the **VAR_ACC_ADDR** structure, pointed to by **addr**.

read_ind_fun

Usage: This user-supplied function is called by MMS-EASE after the virtual machine has resolved the address of a variable by calling **u_get_named_addr** during reads to named variables handled by the virtual machine. It is pointed to by the **read_ind_fun** member of the named variable definition structure, **NAMED_VAR**. Because the **u_get_named_addr** function is called ANY time read access is made to ANY named variable, it does not allow dealing with a specific variable differently than others of the same type without considerable programming. By changing the appropriate member of the **NAMED_VAR** structure that points to the read indication function, a specific variable can be treated differently than the rest of the variables of the same type by having MMS-EASE call this function before generating the read response. Upon returning from this function with no error, MMS-EASE converts the data from the pointer location into the read response PDU being sent. If a null pointer is returned, MMS-EASE sends an error response.

Function Prototype:

```
ST_CHAR *(*read_ind_fun)(ST_CHAR *src, ST_INT len);
```

Inputs:

`src` This pointer to the data is returned by the **u_get_named_addr** function. It should be equal to the address of where the local variable is stored.

`len` This is the length of the data specified in the named type definition structure (**NAMED_TYPE**) member **blocked_len**. It is equal to the number of bytes the virtual machine will read.

Return Value:

`ST_CHAR *` This function should return a character pointer to the data that is to be read. In case of an error, the pointer is set to null and **mms_op_err** is written with the error code.

NOTE: THIS FUNCTION IS COMPLETELY OPTIONAL; IN MOST CASES, IT CAN BE IGNORED. If the variable exists in local memory, a function for (***read_ind_fun**) will not need to be written. MMS-EASE automatically places a pointer to a default function to handle variables existing in local memory into the **NAMED_TYPE** structure when the variable type is defined.

However, if a variable type is defined that does not exist in local memory, a pointer MUST be supplied to this function in the **NAMED_VAR** or **NAMED_TYPE** structure. The appropriate (***read_ind_fun**) function must be written to handle that variable if read responses for this variable are to be sent using virtual machine. If a new type is defined, the (***read_ind_fun**) function is written and there is a pointer to it in the **NAMED_TYPE** table. When a variable name is added using **ms_add_named_var**, MMS-EASE automatically places the pointer from the **NAMED_TYPE** structure into the **NAMED_VAR** structure.

write_ind_fun

Usage: This user-supplied function is called by MMS-EASE after the virtual machine has resolved the address of a variable by calling **u_get_named_addr** during writes to named variables handled by the virtual machine. This is pointed to by the **write_ind_fun** member of the variable definition structure, **NAMED_VAR**. It is responsible for actually performing the write of the variable before the write response being sent. By changing the appropriate member of the **NAMED_VAR** structure pointing to the write indication function, a specific variable can be treated differently than the rest of the variables of the same type. Or, your own function can be supplied to perform any special processing required before generating the write response. After actually writing the variable and returning from this function with no error, MMS-EASE sends a write response PDU. If an error is returned, MMS-EASE sends an error response.

Function Prototype:

```
ST_INT (*write_ind_fun)(ST_CHAR *src,
                        ST_CHAR *dest,
                        ST_INT len);
```

Parameters:

`src` This pointer to the data is supplied in the write indication PDU already translated from ASN.1 into the local blocked format.

`dest` This pointer to the address of the local variable was returned by the **u_get_named_addr** function.

`len` This is the length of the data specified in the named type definition structure (**NAMED_TYPE**) member **blocked_len**. It is equal to the number of bytes to write.

Return Value: `ST_RET` **SD_SUCCESS**. No Error.

<> `0`. Error Code.

NOTE: THIS FUNCTION IS COMPLETELY OPTIONAL; IN MOST CASES, IT CAN BE SIMPLY IGNORED. If the variable exists in local memory, a function for (***write_ind_fun**) does not need to be written. MMS-EASE places a pointer to a default function to handle variables existing in local memory into the **NAMED_TYPE** structure when the variable type is defined.

However, if a variable is defined that does not exist in local memory, a pointer to this function MUST be supplied in the **NAMED_VAR** or **NAMED_TYPE** structure. The appropriate (***write_ind_fun**) function MUST be written to actually perform the writes for that variable if write responses for that variable are sent using the virtual machine. If a new type is defined, the (***write_ind_fun**) function is written and a pointer to it in the **NAMED_TYPE** table. This occurs when a variable name is added using **ms_add_named_var**. MMS-EASE automatically places the pointer from the **NAMED_TYPE** structure into the **NAMED_VAR** structure.

Variable Access Logging Support Functions

The functions below can be called directly to log an Alternate Access or a runtime type. These functions use the MMS-EASE **S_LOG** system to log. No masks are used internally.

ms_log_alt_access

Usage: This support function is used to log information on an Alternate Access type using SLOG.

Function Prototype: `ST_VOID ms_log_alt_access (ALT_ACCESS *alt_acc);`

Parameters:

`alt_acc` This pointer to an **`ALT_ACCESS`** structure contains the alternate access information corresponding to the data to be logged.

Return Value: `ST_VOID` (ignored)

ms_log_runtime

Usage: This support function is used to log information on an Alternate Access type using SLOG.

Function Prototype: `ST_VOID ms_log_runtime (RUNTIME_TYPE *rt, ST_INT num_rt);`

Parameters:

`rt`	This pointer to a **`RUNTIME_TYPE`** structure contains the runtime type information corresponding to the data to be logged.
`num_rt`	This indicates the number of runtime types present in this data.

Return Value: `ST_VOID` (Ignored)

Miscellaneous Variable Access Support Functions

There are three support functions in the MMS-EASE VMI that can be called after a Variable Access indication has been received. These are used to help processing request arguments. The function **ms_extract_varname** can be used to determine what variable names that a Read, Write, or InformationReport indication refer to. The functions **ms_extract_write_data** and **ms_extract_info_data** can be used to put the data from a Write or InformationReport indication into specified memory locations for manipulation by the local user application.

The support functions **ms_extract_write_data** and **ms_extract_info_data** are specific to each respective service (Write and InformationReport). They are explained in the appropriate sections on the Write Service and InformationReport service respectively. The **ms_extract_varname** is more general and can be used with any of the Read, Write, and InformationReport services. This function is explained on the following page.

ms_extract_varname

Usage: This support function extracts the variable name corresponding to the ith variable (where **i** is the index) from an InformationReport, Read, or Write indication PDU. Use this function to find out which variables are specified in the indications dealing with variables. Once the name of the variable is known, the other virtual machine services can be used to manipulate the variable or read in the data corresponding to this variable. See **ms_asn1_to_locl** or **ms_locl_to_asn1** functions for help in converting variable data. Only use this function for indications containing lists of named variables. It cannot be used for named variable lists or unnamed variables.

Function Prototype:

```
OBJECT_NAME *ms_extract_varname (ST_CHAR *reqinfo,
                                 ST_INT op,
                                 ST_INT indx);
```

Parameters:

`reqinfo` This pointer to the operation-specific data structure corresponds to the particular service (Read, Write, or InformationReport) for which this function is being used to extract variable names. You should cast a pointer to one of the following structures here: **READ_REQ_INFO**, **WRITE_REQ_INFO**, or **INFO_REQ_INFO**.

`op` This opcode indicates what type of service (Read, Write, or InformationReport) for which this function is being used to extract variable names. Set this equal to one of the following constants (defined in **mmsop_en.c**): **MMSOP_READ**, **MMSOP_WRITE**, or **MMSOP_INFO_RPT** to indicate which service.

`indx` This index indicates the variable (out of the list of variables contained in the PDU) for which the variable name is to be extracted. If this parameter equals 9, then the 10th variable name is extracted. The first variable in the list of variables would have an index of zero (0).

Return Value:

`OBJECT_NAME *` This pointer to the object name structure, within the operation-specific data structure, is pointed to by **reqinfo**. It contains the ***i***th variable name. See **Volume 1 — Module 2 — MMS Object Name Structure** for more information on this structure. In case of an error, the pointer is set to null and **mms_op_err** is written with the error code.

ms_init_va_size

Usage: This support function is used in conjunction with **ms_va_size** to calculate the size in bytes of Request and Response PDUs for Read, Write, and InformationReport services. The application calls this function to supply information that initializes the size calculations. A typical use of this function is to determine how many variables to send in a Request without exceeding the negotiated MMS segment size. Fitting as many variables as possible in a variable access leads to the optimal throughput of transaction objects over the network .

There is more than one way to produce the ASN.1 encoding for a MMS PDU. This function calculates the length of the ASN/.1 encodings based on knowing the vendor of the ASN.1 encode/decode algorithms. MMS-EASE tends to produce optimal encodings by packing the most amount of information in the least amount of space. Because the lengths of the PDU may change according to the MMS implementation, the following global variables are supplied to calculate best and worst case encoding calculations:

ms_req_bld_id This is the vendor supplying the ASN.1 tools used to encode the MMS Request. This may be set to the constant **MMS_EASE_BUILDER** or **UNKNOWN_BUILDER**.

ms_resp_bld_id This is the vendor supplying the ASN.1 tools used to encode the MMS Response. This may be set to the constant **MMS_EASE_BUILDER** or **UNKNOWN_BUILDER**.

```
#define MMS_EASE_BUILDER  0
#define UNKNOWN_BUILDER         1
```

Function Prototype:

```
ST_RET ms_init_va_size (ST_INT chan,
                        ST_UCHAR spec_in_rslt,
                        ST_INT16 var_acc_tag,
                        OBJECT_NAME *vl_name,
                        ST_LONG *req_size_out,
                        ST_LONG *resp_size_out);
```

Input Parameters:

`op` This contains the type of VariableAccess service; possible values are **MMSOP_READ**, **MMSOP_WRITE**, and **MMSOP_INFO_RPT**.

`spec_in_rslt` This is used with the Read service to calculate the extra overhead associated with sending the SpecificationinResult parameter.

`var_acc_tag` This is used to determine the overhead associated with a ListOfVariables as compared with a NamedVariableList. Valid arguments are **VAR_ACC_VARLIST** or **VAR_ACC_NAMEDLIST**.

. . . continued on the following page. . .

ms_init_va_size . . . cont'd from preceding page . . .

Input Parameters (cont'd):

`vl_name` This is a pointer to a structure of type **`OBJECT_NAME`** that is used to calculate the overhead associated with a service involving a NamedVariableList. See **Volume 1 — Module 2 — MMS Object Name Structure** for more information on this structure.

Output Parameters:

`req_size_out` This is an output parameter that contains the number of bytes of overhead assoicated with the service Request PDU.

`resp_size_out` This is an output parameter that contains the number of bytes of overhead associated with the service Response PDU.

Return Value: `ST_RET` **`SD_SUCCESS`**. No error.

`SD_FAILURE`. Error.

ms_va_size

Usage: This support function is used in conjunction with **ms_init_va_size** to calculate the size in bytes of Request and Response PDUs for Read, Write, and InformationReport services. The application calls this function for each variable in **VARIABLE_LIST**, and after the function returns, the **req_size_out**, and **resp_size_out** may be compared to the MMS segment size to see if the variable will fit in the PDU. A typical use for this function is to calculate how many variables will fit into the PDU before sending the MMS request. Fitting as many variables as possible in a variable access leads to the optimal throughput of transaction objects over the network .

Function Prototype:

```
ST_RET ms_va_size (ST_INT op,
                   ST_UCHAR spec_in_rslt,
                   ST_INT16 var_acc_tag,
                   VARIABLE_LIST *vl,
                   NAMED_TYPE *type,
                   ST_LONG *req_size_out,
                   ST_LONG *resp_size_out);
```

Input Parameters:

`op` This opcode indicates what type of service (Read, Write, or InformationReport) for which this function is being used to extract variable names. Set this equal to one of the following constants (defined in **mmsop_en.c**): **MMSOP_READ**, **MMSOP_WRITE**, or **MMSOP_INFO_RPT** to indicate which service.

`spec_in_rslt` This is used when **op = MMSOP_READ** and indicates if the size estimation should account for SpecificationinResult being included in the MMS Read Response.

`var_acc_tag` This determines if the VariableAccess is for a ListOfVariables as compared with a NamedVariableList. Valid arguments are **VAR_ACC_VARLIST** or **VAR_ACC_NAMEDLIST**.

`vl_name` This pointer of structure type **VARIABLE_LIST** contains the VariableSpecification for a variable.

`type` This pointer to the **NAMED_TYPE** structure contains the type information associated with a variable.

Output Parameters:

`req_size_out` This is a pointer to the size of the request PDU in bytes. The additional size required by the variable will be added to the value supplied as an input.

`resp_size_out` This is a pointer to the size of the response PDU in bytes. The additional size required by the variable will be added to the value supplied as an input.

Return Value: `ST_RET` **SD_SUCCESS**. No error.

SD_FAILURE. Error.

Virtual Machine Request And Response Functions

There are several other Virtual Machine functions that do not just manipulate the Virtual Machine Database or provide miscellaneous support functions. The Virtual Machine request and response functions generate requests and responses for various Variable Access services. They access the Virtual Machine Database, and thereby present a simple interface to the user application.

mv_read	mv_write	mv_info
mv_readvars	mv_writevars	mv_infovars
mv_read_variables	mv_write_variables	mv_info_report
mv_read_resp	mv_write_resp	mv_info_nvlist
mv_read_response	mv_namelist_resp	
mv_defvar_resp	mv_defvlist_resp	mv_deftype_resp
mv_delvar_resp	mv_delvlist_resp	mv_deltype_resp
mv_getvar_resp	mv_getvlist_resp	mv_gettype_resp

In addition to the Virtual Machine request functions, the corresponding user confirmation functions are listed below:

u_mv_read_conf	u_mv_write_conf
u_mv_read_vars_conf	u_mv_write_vars_conf

Levels of Complexity

Note that for the Read, Write, and Information Report services, there are several request functions, and several response functions. That is because there are different levels of complexity offered by the MMS-EASE interface. At one extreme is the PPI. This can support any features of the MMS Variable Access, but it is difficult to use. Above that are the functions **mv_read_variables**, **mv_write_variables**, and **mv_info_report**. These have most of the functionality of the PPI, but also provide many of the value-added features of the VMI (such as automatic encoding of data between the local representation and the ASN.1-encoded format). The next easiest functions to use are **mv_readvars**, **mv_writevars**, and **mv_infovars**. These allow access to one or more remote variables (no described variables). Finally, the easiest functions in the VMI to use (regarding the Read, Write, and Information Report services) are **mv_read**, **mv_write**, and **mv_info**. These allow one named variable at a time to be accessed.

It is strongly recommended to use the Virtual Machine functions, at some level, when possible. Variable access through the PPI is quite involved, and will result in considerably longer and more complex code.

Interdependence of VMI Functions

Because the VMI Request and Response functions reference and use the Virtual Machine Database, they require the use of the VMI Database Management functions. These are the **ms_** support functions already covered for configuring the database ahead of time. The **mv_** functions listed here are intimately related to and depend on the **ms_** functions explained earlier. To see how interrelated the Virtual Machine Request and Response functions are with the **ms_** support functions, let us look at some sample scenarios:

EXAMPLE 1:

As a Server, an application would call **ms_add_std_types** or **ms_add_named_type** one or more times to set up the local types. It would normally call **ms_mk_asn1_type** before each call to **ms_add_named_type**. It would then call **ms_add_named_var** one or more times to set up all the local named variables. When **u_read_ind** is called, indicating that a remote application is requesting to read one or more variables, the local application need only call **mv_read_resp**. MMS-EASE will get the values and respond automatically. In the process of getting the data values, the VMI calls the address resolution functions, **u_get_named_addr** and **read_ind_fun**.

EXAMPLE 2:

As a Client, an application also would have to set up its local types. Before reading a list of remote named variables, it would prepare an array of **mv_vardesc** structures. Each would have to specify the name of a remote variable, the name of a local type, and the location in memory to put the resulting data. Then the request function **mv_readvars** is called to send out the Read request. At some point later when the confirm is received, **u_mv_read_conf** is called by MMS-EASE, and the application can manipulate the data in the specified memory locations. All the conversion to local representation is already done when the confirm function is called. The application can immediately do real work with the data from the remote variables.

NOTE: The **MV_VARDESC** structure is a common structure used by several Virtual Machine request functions, and so its definition is given in this section. All other documentation regarding the Virtual Machine request and response functions is presented under the appropriate services.

Variable List Data Structure

The following data structure is used by the virtual machine when handling lists of variables using the **mv_readvars**, **mv_writevars**, or the **mv_infovars** functions. This structure contains the actual list of named variables.

```
struct mv_vardesc
  {
  OBJECT_NAME name;
  OBJECT_NAME type;
  ST_CHAR     *data;
  };
typedef struct mv_vardesc MV_VARDESC;
```

Fields:

`name` This structure of type **OBJECT_NAME** contains the name of the remote variable to be read or written.

`type` This structure of type **OBJECT_NAME** contains the name of the locally defined named type corresponding to the variable type on the remote node. This type must be defined locally. Although this type need not be defined on the remote node, it must match the actual variable type being read or written.

`data` This is a pointer to local memory where the results of a Read request should go, or where to obtain the data from in the case of a Write or InformationReport operation.

NOTE: See **Volume 1 — Module 2 — MMS Object Name Structure** for a detailed description of the **OBJECT_NAME** structure.

3. Paired Primitive Interface

Variable Access Support Structures

This section illustrates the various data structures used for variable access at the PPI level in MMS-EASE. Normally the virtual machine provides a simpler mechanism for dealing with variables. These structures will not need to be used for most of the virtual machine functions. Regardless, in order to understand fully this section, you must be familiar with the MMS specification and how it describes variables. The various structure members are described by using descriptions corresponding to the MMS specification.

Address Structures

UNCONST_ADDR VAR_ACC_ADDR

These structures are used to describe the address of variables. Addresses are always implementation-specific and are not standardized. There are three forms that MMS addresses can take on, but their meanings and use are left for the various vendors of MMS hardware and software to specify.

```
struct unconst_addr
  {
  ST_INT    unc_len;
  ST_UCHAR   *unc_ptr;
  SD_END_STRUCT
  };
typedef struct unconst_addr UNCONST_ADDR;
```

Fields :

unc_len This is the length of the unconstrained address pointed to by **unc_ptr**.

unc_ptr This pointer to the unconstrained address is stored as an OctetString.

An unconstrained address is just as the name implies: the address can contain any information at all. An unconstrained address is used when a relative (numeric) or symbolic address is not suitable.

```
struct var_acc_addr
  {
  ST_INT16 addr_tag;
  union
    {
    ST_UINT32    num_addr;
    ST_CHAR       *sym_addr;
    UNCONST_ADDR unc_addr;
    } addr;
  };
typedef struct var_acc_addr VAR_ACC_ADDR;
```

Fields:

addr_tag This is a tag indicating the type of address:

NUM_ADDR. This represents the numeric address. Used with the **num_addr** member of **addr**.

SYM_ADDR . This represents the symbolic address. Use the **sym_addr** member of **addr**.

UNCON_ADDR . This represents the unconstrained address. Use **unc_addr** member of **addr**.

`num_addr` This contains the numeric address of the variable. Used if **`addr_tag = NUM_ADDR`**.

`sym_addr` This pointer to the symbolic address of the variable is used if **`addr_tag = SYM_ADDR`**.

`unc_addr` This structure of type **`UNCONST_ADDR`** contains the unconstrained address of the variable. Used if **`addr_tag = UNCON_ADDR`**.

Variable Access Result Structures

The following describes the data structures used to represent the results of a variable access including success or failure information and a variable's data.

`VAR_ACC_DATA`

This structure is used to hold the data that was the result of a successful variable access.

```
struct var_acc_data
  {
  ST_INT   len;
  ST_UCHAR *data;
  };
typedef struct var_acc_data VAR_ACC_DATA;
```

Fields:

`len` This is the length, in bytes, of the data pointed to by **`data`**.

`data` This is a pointer to the ASN.1 encoded data resulting from the successful variable access. The data contained in this buffer must conform to the ASN.1 encoding rules. It also must conform to the following ASN.1 syntax as specified by ISO 9506 (the MMS IS specification). This is explained below.

```
Data ::= CHOICE {
context tag 0 is reserved for access_result
   array              [1]  IMPLICIT SEQUENCE OF Data,
   structure          [2]  IMPLICIT SEQUENCE OF Data,
   boolean            [3]  IMPLICIT BOOLEAN,
   bit-string         [4]  IMPLICIT BIT STRING,
   integer            [5]  IMPLICIT INTEGER,
   unsigned           [6]  IMPLICIT INTEGER,
   floating-point     [7]  IMPLICIT FloatingPoint,
   real               [8]  IMPLICIT REAL,
   octet-string       [9]  IMPLICIT OCTETSTRING,
   visible-string     [10] IMPLICIT VisibleString,
   generalized-time   [11] IMPLICIT GeneralizedTime
   binary-time        [12] IMPLICIT TimeOfDay,
   bcd                [13] IMPLICIT INTEGER,
   booleanArray       [14] IMPLICIT BITSTRING
   objid              [15] IMPLICIT OBJECT IDENTIFIER
}
```

Refer to the MMS IS specification. The data found in this element must conform to a particular type found in the type specification for this variable. See the following description of **`VAR_ACC_TSPEC`**. The virtual machine should be used for variable access since it automatically performs the translation of this data into the appropriate local variables. This eliminates having to deal with the above. See the section on the virtual machine variable access starting on page 2-109 for more information.

If the PPI is used, some of the virtual machine functions can be used to encode and decode ASN.1 Data as long as the type definition has been entered into the virtual machine database. See the **`ms_locl_to_asn1`** and **`ms_asn1_to_locl`** support functions starting on page 2-109 for more information.

ACCESS_RESULT

This structure specifies the results of a data access. It may contain the actual data resulting from a Read, the data to be written during a Write, or error information regarding the failure of the variable access.

```
struct access_result
  {
  ST_INT16     acc_rslt_tag;
  ST_INT16     failure;
  VAR_ACC_DATA va_data;
  };
typedef struct access_result ACCESS_RESULT;
```

Fields:

`acc_rslt_tag` This is a tag indicating the result of the variable access:

ACC_RSLT_FAILURE. Access failed. See **failure** member below.

ACC_RSLT_SUCCESS. Access Succeeded. See **va_data** member below.

`failure` This indicates the reason for failure of the access. Used if **acc_rslt_tag = ACC_RSLT_FAILURE**.

ARE_OBJ_INVALIDATED. An attempted access references a defined object that has an undefined reference attribute. This represents a permanent error for access attempts to that object.

ARE_HW_FAULT. An attempt to access the variable has failed due to a hardware fault.

ARE_TEMP_UNAVAIL. The requested variable is temporarily unavailable for the requested access.

ARE_OBJ_ACCESS_DENIED. The MMS Client has insufficient privilege to request this operation.

ARE_OBJ_UNDEFINED. The object with the desired name does not exist.

ARE_INVAL_ADDR. Reference to the unnamed variable object's specified address is invalid because the specified format is incorrect or is out of range.

ARE_TYPE_UNSUPPORTED. An inappropriate or unsupported type is specified for a variable.

ARE_TYPE_INCONSISTENT. A type is specified that is inconsistent with the service or referenced object.

ARE_OBJ_ATTR_INCONSISTENT. The object is specified with inconsistent attributes.

ARE_OBJ_ACC_UNSUPPORTED. The variable is not defined to allow requested access.

ARE_OBJ_NONEXISTENT. The variable is non-existent.

`va_data` This structure of type **VAR_ACC_DATA** contains the data for this variable if **acc_rslt_tag = ACC_RSLT_SUCCESS**.

Variable Type Structure

VAR_ACC_TSPEC

This structure is used to define the type of a particular variable. This type definition is the same as what is used by the virtual machine.

```
struct var_acc_tspec
  {
  ST_INT   len;
  ST_UCHAR *data;
  };
typedef struct var_acc_tspec VAR_ACC_TSPEC;
```

Fields:

`len` This is the length, in bytes, of the data pointed to by **`data`**.

`data` This is a pointer to the ASN.1 encoded type definition for the variable being accessed. The data contained in this buffer must conform to the ASN.1 encoding rules and to the ASN.1 syntax as specified by the MMS specification. Please refer to the section of this manual containing information on the Type Specification in **Volume 1 — Appendix I** for more information.

If using the VMI for handling variable access, you do not need to deal with this data structure. Even if the PPI is used for variable access, you may still want to create a virtual machine type definition. This is so that the **`ms_locl_to_asn1`** and **`ms_asn1_to_locl`** support functions can be used to convert data to and from ASN.1. Once a virtual machine type is defined, the data pointed to by **`named_type.asn1ptr`** can be used, and placed in the **`VAR_ACC_TSPEC`** structure for Paired Primitive Interface operations.

Described Variable Structure

VARIABLE_DESCR

This structure is used when access is made to a described variable. Described variable access specifies the type and address of the variable each timze that variable is accessed. This is different from named variables where access can be made on the name alone, and other unnamed variables where access can be made on address alone.

```
struct variable_descr
  {
  VAR_ACC_ADDR      address;
  VAR_ACC_TSPEC     type;
  };
typedef struct variable_descr VARIABLE_DESCR;
```

Fields:

`address` This structure of type **`VAR_ACC_ADDR`** contains this variable's address.

`type` This structure of type **`VAR_ACC_TSPEC`** contains this variable's type definition.

Variable Specification Structure

VARIABLE_SPEC

This structure is used to hold a variable specification. When this structure and all its sub-structures are filled out completely, it specifies the variable being accessed. It contains information about whether the variable is named, addressed, or described. It is used during PPI variable access operations. Please note that this structure calls out the use of several previously documented structures.

```
struct variable_spec
  {
  ST_INT16 var_spec_tag;
  union
    {
    OBJECT_NAME      name;
    VAR_ACC_ADDR     address;
    VARIABLE_DESCR   var_descr;
    SCATTERED_ACCESS sa_descr;
    } vs;
  };
typedef struct variable_spec VARIABLE_SPEC;
```

Fields:

`var_spec_tag` This is a value indicating the type of variable:

VA_SPEC_NAMED. Access variable by name only.
VA_SPEC_ADDRESSED. Access variable by address only.
VA_SPEC_DESCRIBED. Access variable by address and type.
VA_SPEC_SCATTERED. Scattered Access.

VA_SPEC_INVALIDATED. Invalidated Variable. Used during responses only when the specification of the variable is to be returned in the response to a variable access request. An invalidated variable object occurs when access to a scattered access object is attempted where one or more of the underlying objects (defined as a part of the accessed scattered access object) has been deleted.

`name` This structure of type **OBJECT_NAME** contains the name of the variable when the variable is to be accessed by name only. Used if **var_spec_tag = VA_SPEC_NAMED**.

`address` This structure of type **VAR_ACC_ADDR** contains the address of the variable when the variable is to be accessed by addressed only. Used if **var_spec_tag = VA_SPEC_ADDRESSED**.

`var_descr` This structure of type **VARIABLE_DESCR** contains the description of the variable if the variable is to be accessed by specifying the address and type. Used if **var_spec_tag = VA_SPEC_DESCRIBED**.

`sa_descr` This structure of type **SCATTERED_ACCESS** contains the scattered access description of the variable. Used if **var_spec_tag = VA_SPEC_SCATTERED**.

Variable List Structure

VARIABLE_LIST

This structure is used to specify a variable and any alternative access on that variable in the list of variables to be accessed.

```
struct variable_list
  {
  VARIABLE_SPEC    var_spec;
  ST_BOOLEAN       alt_access_pres;
  ALTERNATE_ACCESS alt_access;
  };
typedef struct variable_list VARIABLE_LIST;
```

Fields:

`var_spec` This structure of type **VARIABLE_SPEC** contains the variable specification for this element of the variable list.

`alt_access_pres` **SD_TRUE**. **alt_access** is present

SD_FALSE. **alt_access** is not present

`alt_access` If used, this structure of type **ALTERNATE_ACCESS** contains the alternate access description. See the next page for more information on this structure.

Variable Access Specification Structure

VAR_ACC_SPEC

This structure is used to specify everything needed for a particular variable access operation. It is used in nearly all the operation-specific data structures for the variable access services of the PPI. Nearly all previously documented PPI variable access support structures are used in one way or another inside the sub-structures of this master structure.

```
struct var_acc_spec
  {
  ST_INT16  var_acc_tag;
  struct     object_name       vl_name;
  ST_INT     num_of_variables;
/*struct     variable_list       var_list [num_of_variables];           */
  SD_END_STRUCT
  };
typedef struct var_acc_spec VAR_ACC_SPEC;
```

Fields:

`var_acc_tag` This is a value indicating the type of access. Options are:

VAR_ACC_VARLIST. List of Variables
VAR_ACC_NAMEDLIST. Named Variable List

`vl_name` This structure of type **OBJECT_NAME** contains the name of this Named Variable List. Used if **var_acc_tag = VAR_ACC_NAMELIST**.

`num_of_variables` This indicates the number of variables in this list if this access is for a list of of variables. Used if **var_acc_tag = VAR_ACC_VARLIST**.

NOTE: To read a single variable, you would read a list of one (e.g., **num_of_variables = 1**).

`var_list` This array of structures of type **`VARIABLE_LIST`** contains the variable descriptions for the list of variables to be accessed. Used if **`var_acc_tag = VAR_ACC_VARLIST`**.

NOTE: When allocating Operation-Specific data structures containing a structure of type **`VAR_ACC_SPEC`**, make sure that sufficient memory is allocated to hold the list of variables contained in **`var_list`**. See the specific PPI data structure for more information on how to allocate this memory (under the Read, Write, and InformationReport sections).

Scattered Access Structure

`SCATTERED_ACCESS`

This structure is used to hold the ASN.1 encoding for scattered access. Scattered access is currently not supported by the VMI. However, for those knowledgeable in ASN.1 and MMS, this option can be used by encoding the appropriate ASN.1 into this structures when using the PPI.

Please refer to the MMS specification for more detail on the ASN.1 representation of the scattered access object.

```
struct scattered_access
  {
  ST_INT len;
  ST_UCHAR *data;
  };
typedef struct scattered_access  SCATTERED_ACCESS;
```

Fields:

`len` This is the length, in bytes, of the scattered access description pointed to by **`data`**.

`data` This is a pointer to data that contains the scattered access description.

Alternate Access Structure

`ALTERNATE_ACCESS`

This structure is used to hold the ASN.1 encoding for alternate access. Alternate access is supported for the VMI and it is recommended to use the VMI instead of the PPI. However, for those knowledgeable in ASN.1 and MMS, this option can be used by encoding the appropriate ASN.1 into this structure when using the PPI. Please refer to the MMS specification for more detail on the ASN.1 representation of alternate access objects.

An alternate Access description specifies an alternative view of a variable's type (the abstract syntax and the range of possible values of a real variable). It can be used to alter the perceived abstract syntax (using MMS services) or to restrict access to a subset of a range of possible values (partial access), or both.

```
struct alternate_access
  {
  ST_INT   len;
  ST_UCHAR *data;
  };
typedef struct alternate_access  ALTERNATE_ACCESS;
```

Fields:

`len` This is the length, in bytes, of the alternate access description pointed to by **`data`**.

`data` This is a pointer to data that contains the alternate access description.

4. Read Service

This service is used by a Client application to request that a Server VMD return the value of one or more variables defined at the VMD. MMS-EASE provides both Client and Server functionality.

As mentioned on page 2-85, there are several levels of complexity provided in MMS-EASE for the Read service. The PPI is the most complex and requires the most work by the user application. The easiest request functions to use are **mv_read** and **mv_readvars**, together with their corresponding confirm function, **u_mv_read_conf**. These functions will read named variables only. The request function **mv_read_variables**, and its confirm function **u_mv_read_vars_conf**, provide almost as much power as the PPI (including reading described variables). Therefore, they are somewhat more complex than **mv_read** and **mv_readvars**. The VMI response functions **mv_read_resp** and **mv_read_response** are much easier to use than the PPI. However, they require a Virtual Machine Database to have been configured ahead of time (to different degrees). All these functions will be described in this chapter.

Primitive Level Read Operations

The following section contains information on how to use the paired primitive interface for the Read service. It covers available data structures used by the PPI, and the four primitive level functions that together make up the Read service. See **Volume 1 — Module 2 — General Sequence of Events — Confirmed Services** for a detailed explanation of the sequence of events that occur within the MMS-EASE environment during confirmed services.

- The Read service consists of the paired primitive functions of **mp_read**, **u_read_ind**, **mp_read_resp**, and **u_mp_read_conf**.

Data Structures

Request/Indication

The operation-specific data structure described below is used by the Client in issuing the variable read request function (**mp_read**). It is received by the Server when a variable read indication function (**u_read_ind**) is received.

```
struct read_req_info
  {
  ST_BOOLEAN    spec_in_result;
  VAR_ACC_SPEC  va_spec;
/*VARIABLE_LIST var_list [va_spec.num_of_variables];             */
/*SD_END_STRUCT                                                      */
  };
typedef struct read_req_info READ_REQ_INFO;
```

Fields:

`spec_in_result` — **SD_FALSE**. Do not include the access specification in the response. This is the default.

SD_TRUE. Include the access specification (the type and address information) in the response.

`va_spec` — This structure of type **VAR_ACC_SPEC** contains the variable access specification.

`var_list` — This array of structures of type **VARIABLE_LIST** “includes” a list of variables to be read.

NOTES:

1. See the description of the variable access data structures starting on page 2-150 for more information on the **VARIABLE_LIST** and **VAR_ACC_SPEC** structures.
2. FOR REQUEST ONLY, when allocating a data structure of type **READ_REQ_INFO**, enough memory must be allocated to hold the information for the **var_list** member of the structure. The following C statement can be used:

```
info = (READ_REQ_INFO *) chk_malloc(sizeof(READ_REQ_INFO) + (num_of_variables *
            sizeof(VARIABLE_LIST)));
```

Response/Confirm

The operation-specific data structure described on the following page is used by the Server in issuing a variable read response function (**mp_read_resp**). It is received by the Client when a variable read confirm function (**u_mp_read_conf**) is received.

```
struct read_resp_info
  {
  ST_BOOLEAN    va_spec_pres;
  ST_INT         num_of_acc_result;
  ACCESS_RESULT *acc_rslt_list;
  VAR_ACC_SPEC  va_spec;
/*VARIABLE_LIST var_list [va_spec.num_of_variables];             */
/*ACCESS_RESULT acc_result_list [num_of_acc_result];             */
/*SD_END_STRUCT                                                  */
  };
typedef struct read_resp_info READ_RESP_INFO;
```

Fields:

va_spec_pres **SD_FALSE**. Do Not include the **va_spec** in the PDU.

SD_TRUE. Include **va_spec** in the PDU. This should only be included if the variable access specification was requested in the Read indication, **read_req_info.spec_in_result <> 0**.

num_of_acc_result This indicates the number of access results in the array of structures pointed to by **acc_rslt_list**.

acc_rslt_list This pointer to **acc_result_list** is an array of structures of type **ACCESS_RESULT** containing the information read.

va_spec This structure of type **VAR_ACC_SPEC** contains the variable access specification information.

var_list This array of structures of type **VARIABLE_LIST** contains the list of variables to be included in this PDU.

acc_result_list This array of structures of type **ACCESS_RESULT** contains the read results for the specified variables.

NOTES:

1. See the description of the variable access data structures starting on page 2-150 for more information on the **ACCESS_RESULT**, **VAR_ACC_SPEC**, and **VARIABLE_LIST** structures

2. FOR RESPONSE ONLY, when allocating a data structure of type **READ_RESP_INFO**, enough memory must be allocated to hold the information for the **var_list** and **acc_result_list** members of the structure. The following C statement can be used:

```
info = (READ_RESP_INFO *) chk_malloc(sizeof(READ_RESP_INFO) + (num_of_acc_result
            * sizeof(ACCESS_RESULT)) +      (num_of_variables *
            sizeof(VARIABLE_LIST)));
```

Paired Primitive Interface Functions

mp_read

Usage: This primitive request function sends a Read request PDU. It uses the data from a structure of type **READ_REQ_INFO**, pointed to by **info**. This function is used to read the contents of a variable or variables located at a remote node.

Function Prototype: `MMSREQ_PEND *mp_read (ST_INT chan, READ_REQ_INFO *info);`

Parameters:

`chan`	This integer contains the channel number over which the Read PDU is to be sent.
`info`	This pointer to an Operation-Specific data structure of type **READ_REQ_INFO** contains information specific to the Read PDU to be sent.

Return Value:

`MMSREQ_PEND *`	This pointer to the request control data structure of type **MMSREQ_PEND** is used to send the Read PDU. In case of an error, the pointer is set to null and **mms_op_err** is written with the error code.

Corresponding User Confirmation Function: **u_mp_read_conf**

Operation-Specific Data Structure Used: **READ_REQ_INFO**

See page 2-153 for a detailed description of this structure.

u_read_ind

Usage: This user function is called when a Read indication is received by a Server node. The contents and operation of this function are user-defined and depend on the application requirements. To conform to the MMS protocol requirements, the application must do one of the following some time after the indication is received:

1) send a positive response using either:

   a) the primitive response function (**`mp_read_resp`**) after reading the specified variable(s),

   b) a virtual machine response function (**`mv_read_resp`** or **`mv_read_response`**) to have the virtual machine take the requested action and send the response, or

2) call the appropriate primitive negative (error) response function (**`mp_err_resp`**). See **Volume 3 — Module 11 — MMS-EASE Error Handling** for an explanation of this function.

See **Volume 1 — Module 1 — User Indication Function Class** for additional information regarding confirmed service indication handling.

Function Prototype: `ST_VOID u_read_ind (MMSREQ_IND *ind);`

Parameters:

`ind` This pointer to the **`MMSREQ_IND`** structure is described in **Volume 1— Module 2 — Request and Indication Control Data Structures**. This pointer must be used to send the response to this indication. The primary item of interest in this structure is **`req->req_info_ptr`**. This is a pointer to the operation-specific data structure for a Read indication (**`READ_REQ_INFO`**). This pointer will always be valid when **`u_read_ind`** is called.

Return Value: `ST_VOID` (ignored)

Operation-Specific Data Structure Used: **`READ_REQ_INFO`**

See page 2-153 for a detailed description of this structure.

mp_read_resp

Usage: This primitive response function sends a positive Read response PDU. This function should be called only after the **u_read_ind** function is called (a Read indication is received), and after the specified variable data has been successfully obtained.

Function Prototype:

```
ST_RET mp_read_resp (MMSREQ_IND *ind,
                     READ_RESP_INFO *info);
```

Parameters:

`ind` This pointer to the indication control structure of type **MMSREQ_IND** is received in the **u_read_ind** function.

`info` This pointer to an Operation-Specific data structure of type **READ_RESP_INFO** contains information specific to the PDU to be sent.

Return Value: `ST_RET`

SD_SUCCESS. No Error.

`<> 0` Error Code.

Corresponding User Indication Function: **u_read_ind**

Operation-Specific Data Structure Used: **READ_RESP_INFO**

See page 2-154 for a detailed description of this structure.

u_mp_read_conf

Usage: This primitive user confirmation function is called when the confirm to a **mp_read** request is received. **resp_err** contains a value indicating whether an error occurred. **resp_info_ptr** may contain data that the remote node sent back in the confirm, or it may contain error information is an error occurred. If no error has occurred (**req->resp_err = CNF_RESP_OK**), read information is available to the application using the **req->resp_info_ptr** pointer. See **Volume 1 — Module 1 — Primitive User Confirmation Function Class** for specific information on the recommended handling of confirms.

Note that the function called for this event may be changed if desired; however, **u_mp_read_conf** is the standard default function.

Function Prototype:

```
ST_VOID u_mp_read_conf (MMSREQ_PEND *req);
```

Parameters:

`req` This pointer to the **MMSREQ_PEND** structure is returned from the original **mp_read** request function. See **Volume 1 — Module 2 — Request and Indication Control Data Structures** for more information on this structure.

For a positive response (**req->resp_err = CNF_RESP_OK**), the request control structure member **resp_info_ptr** is a pointer to the operation-specific data structure (**READ_RESP_INFO**) for the Read function.

For a negative response, or other errors, see **Volume 3 — Module 11 — MMS-EASE Error Handling** for more information.

Return Value: `ST_VOID` (ignored)

Operation-Specific Data Structure Used: **READ_RESP_INFO**

See page 2-154 for a detailed description of this structure.

Virtual Machine Read Operations

The following section contains information on how to use the virtual machine interface for the Read service. It covers available data structures used by the VMI, and the virtual machine functions that together make up the Read service.

Sequence of Events

- The Read service consists of the virtual machine functions of **mv_read**, **mv_readvars**, **mv_read_variables**, **mv_read_resp**, **mv_read_response**, **u_mv_read_conf**, and **u_mv_read_vars_conf**. Note that the primitive indication function, **u_read_ind** is also used with the VMI. This portion of the Read service for the virtual machine has some exceptions to the standard sequence for virtual machine:

1. Node A sends a named variable read request by calling one of the following:
   mv_read allows the ability to read one remotely defined named variable and place its value in local memory using a locally defined named type definition.
   mv_readvars allows reading a **LIST** of remotely defined named variables and placing them in local memory.
   mv_read_variables allows reading a **LIST** of remotely defined variables (named and unnamed) and placing them in local memory. This function allows issuing a request with the flexibility of a primitive level request but with the ability to interpret the results by the virtual machine. Both **MV_READ_REQ_INFO** and **READ_REQ_INFO** structures are used.

   NOTE: If addressed or described variables are used, SISCO recommends that the **mv_read_variables** function be used, since **mv_read** and **mv_readvars** only support named variable access.

2. Within the user-defined **u_read_ind** function on Node B, the user can call either of the response functions: **mv_read_resp**, **or mv_read_response**.

   This rest of this sequence deals with calling **mv_read_resp**.

3. The MMS-EASE on Node B sorts through the operation-specific data corresponding to the read indication received (**READ_REQ_INFO**). It finds the variable and type information corresponding to the variables being read. MMS-EASE then calls the **u_get_named_addr** function and passes the address information found in its database to this function.

4. The user program on Node B determines the variable's physical address, and returns that address from **u_get_named_addr**.

5. MMS-EASE on Node B calls the **read_ind_fun**, pointed to in the **NAMED_VAR** structure, for the variable being read. If the user has modified the **read_ind_fun** from the default, the physical address of the variable must be returned. If the user has not changed the default **read_ind_fun**, no further action is necessary at this step.

6. MMS-EASE on Node B converts the data pointed to by the address returned from **read_ind_fun** from the local C language format into the MMS format. It uses the type definition information found in the virtual machine database for the variable being read. It formats this data into a positive read response PDU. The PDU is sent and a value of SUCCESS is returned to the user's program on Node B that called **mv_read_resp**.

7. If at any time during this processing by the virtual machine, an error is detected, or if a null pointer is returned from either **u_get_named_addr** or **read_ind_fun**, an error response is sent back to Node A. An error code is returned to the user's program on Node B that called **mv_read_resp**.

8. Once the response function returns with a code indicating SUCCESS, the indication is no longer active. The **MMSREQ_IND** structure (used for that indication) is freed up to be used by other indications. If the virtual machine was used for the response and there was an error code, a negative response will have already been sent.

9. At Node A, MMS-EASE decodes the confirm. If the confirm is positive, it automatically converts the returned data into the local representation, and stores it in the memory location(s) specified in the original request. Then the user confirm function, **u_mv_read_conf** or **u_mv_read_vars_conf** is called, depending on which request function is called.

Data Structures

Request/Indication

The data structure described below is used transparently by the Virtual Machine **mv_read** and **mv_readvars** request functions. See pages 2-164 and 2-165 for more information on these functions.

```
struct readvars_req_info
  {
  ST_INT      num_of_vars;
/*MV_VARDESC vardesc_list [num_of_vars];                          */
  SD_END_STRUCT
  };
typedef struct readvars_req_info READVARS_REQ_INFO;
```

Fields:

num_of_vars	This contains the number of variables to be Read in the list of variables in **vardesc_list**.
vardesc_list	This array of structures of type **MV_VARDESC** contains the variable descriptions for all the variables in the list of variables to be read. See page 2-143 for a detailed description of this structure.

NOTE: FOR REQUEST ONLY, when allocating memory for the **READVARS_REQ_INFO** structure, enough memory must be allocated to hold the actual list of variables contained in **vardesc_list**. The following C statement can be used:

```
info = (READVARS_REQ_INFO *) chk_malloc(sizeof(READVARS_REQ_INFO) + (num_of_vars
            * sizeof(MV_VARDESC)));
```

The following data structure is used by the virtual machine function **mv_read_variables**. It gives the power and flexibility of the PPI. A pointer to an array of these structures is passed into the function **mv_read_variables**. See page 2-166 for more information on this function.

```
struct mv_read_req_info
  {
  struct                  /* input (request) information          */
    {
    ST_CHAR     *data_ptr;
    NAMED_TYPE  *type;
    ST_BOOLEAN  alt_acc_pres;
    ST_BOOLEAN  alt_acc_data_packed;
    ALT_ACCESS  alt_acc;
    }i;
  struct                  /* output (result) information          */
    {
    ST_RET      result;
    ST_INT16    err_code;
    ST_INT       num_rt;
    RUNTIME_TYPE *rt_out;
    ST_CHAR      *data_ptr_out;
    }o;
  };
typedef struct mv_read_req_info  MV_READ_REQ_INFO;
```

Input fields:

`data_ptr` If **result = ACC_RSLT_SUCCESS**, this is the pointer to the local memory location where the resulting data for this variable will be put. Please note that a buffer must be committed for the duration of the request.

`type` This pointer to a structure of type **NAMED_TYPE** contains the type associated with this variable. See page 2-62 for more information on this data structure.

`alt_acc_pres` **SD_TRUE**. **alt_acc** is present.

SD_FALSE. **alt_acc** is not present.

`alt_acc_data_packed` **SD_TRUE**. The data present in **alt_acc** is packed.

SD_FALSE. The data present in **alt_acc** is not packed.

`alt_acc` This structure of type **ALT_ACCESS** contains the alternate access description. See page 2-0 for more information on this structure.

Output fields:

`result` This indicates the outcome of the **mv_read_variables** function:

ACC_RSLT_FAILURE
ACC_RSLT_SUCCESS

`err_code` If **result = ACC_RSLT_FAILURE**, then this contains the error code indicating the reason for the failure. Two errors that can occur are:

ERROR_RESPONSE
NUM_VAR_MISMATCH

See Appendix A for a list of additional valid error codes.

`num_rt` This element is used if the type of data is unknown. It indicates the number of runtime types contained in the read.

`rt_out` If the named type is not supplied and the read was a success, the **rt_out** of structure type **RUNTIME_TYPE** points to the local memory location containing the type information derived from the runtime type. This buffer is allocated by MMS-EASE using **chk_calloc**. It must be freed using **chk_free**.

`data_ptr_out` This element is used if the size of data to be read is unknown. If the **`data_ptr`** element is not supplied and the read is a success, the **`data_ptr_out`** points to the local memory location where the resulting data from this variable will be put. The buffer is allocated by MMS-EASE using **`chk_calloc`**. It must be freed using **`chk_free`**.

Response/Confirm

The following data structure, described on the next page, is used to respond to read indications using the virtual machine function, **`mv_read_response`**. See page 2-169 for more information on this function.

```
struct mv_read_resp_info
  {
  ST_INT16   result;
  ST_INT16   err_code;
  ST_CHAR    *data_ptr;
  NAMED_TYPE *type;
  ST_BOOLEAN alt_acc_pres;
  ST_BOOLEAN alt_acc_data_packed;
  ALT_ACCESS alt_acc;
  };
typedef struct mv_read_resp_info MV_READ_RESP_INFO;
```

Fields:

`result` This indicates the result of the **`mv_read_response`** function:

`ACC_RSLT_FAILURE`

`ACC_RSLT_SUCCESS`

`err_code` If **`result = ACC_RSLT_FAILURE`**, then this contains the error code indicating the reason for the failure. See **Volume 1 — Appendix A** for a list of valid error codes.

`data_ptr` If **`result = ACC_RSLT_SUCCESS`**, this is the pointer to be returned for the variable.

`type` This structure of type **`NAMED_TYPE`** contains the type associated with this variable. See page 2-62 for more information on this data structure.

`alt_acc_pres` **`SD_TRUE`**. **`alt_acc`** is present.

`SD_FALSE`. **`alt_acc`** is not present.

`alt_acc_data_packed` **`SD_TRUE`**. The data present in **`alt_acc`** is packed.

`SD_FALSE`. The data present in **`alt_acc`** is not packed.

`alt_acc` This structure of type **`ALT_ACCESS`** contains the alternate access description. See page 2-0 for more information on this structure.

Virtual Machine Interface Functions

mv_read

Usage: This virtual machine function allows the user to execute the read service without having to deal directly with the operation-specific data structures. This function allows the user to read a remotely defined named variable, and place it in local memory using a locally define named type definition. BE SURE TO SET **m_vmd_select** TO POINT TO THE PROPER VMD STRUCTURE BEFORE CALLING THIS FUNCTION. See **Module 4 — VMD Control** starting on page 2-10 for more information.

Function Prototype:

```
MMSREQ_PEND *mv_read (ST_INT chan,
                      OBJECT_NAME *name,
                      OBJECT_NAME *type,
                      ST_CHAR *dest);
```

Parameters:

`chan` This contains the channel number on which the request is to be sent.

`name` This pointer to a structure of type **OBJECT_NAME** contains the name of the remotely defined named variable whose data is to be read. See **Volume 1 — Module 2 — MMS Object Name Structure** for more information on this structure.

`type` This pointer to a structure of type **OBJECT_NAME** contains the name of a locally defined named type that applies to the variable described by **name**.

`dest` This is a pointer to where the data should be written into local memory when the Read confirm is received.

Return Value:

`MMSREQ_PEND *` This pointer to the request control data structure of type **MMSREQ_PEND** contains information regarding this request. In case of an error, the pointer is set to null and **mms_op_err** is written with the error code.

NOTE: In the request control data structure pointed to by **MMSREQ_PEND**, the **req_info_ptr** member points to a data structure of type **READVARS_REQ_INFO**. This special structure is used transparently by the virtual machine to keep track of the Read request. See page 2-161 for a detailed description of this structure. In most instances, it is not necessary to deal with this structure (**readvars_req_info.num_of_vars** will be equal to 1 for **mv_read**).

mv_readvars

Usage: This virtual machine function allows the user to execute the Read service without having to deal directly with the PPI operation-specific data structures. This function allows the user to read a list of remotely defined named variables, and place them in local memory using locally-defined named type definitions. BE SURE TO SET **m_vmd_select** TO POINT TO THE PROPER VMD STRUCTURE BEFORE CALLING THIS FUNCTION. See **Module 4 — VMD Control** starting on page 2-10 for more information.

Function Prototype:

```
MMSREQ_PEND *mv_readvars (ST_INT chan,
                          ST_INT num_var,
                          MV_VARDESC *info);
```

Parameters:

`chan` This is the channel on which request is to be sent.

`num_var` This indicates the number of variables to be read in this Read request. It must be equal to the number of individual structures of type **MV_VARDESC** pointed to by **info**.

`info` This pointer to the beginning of an array of structures of type **MV_VARDESC** contains the information specifying which variables are to be read. See page 2-143 for more information on this structure.

Return Value:

`MMSREQ_PEND *` This pointer to the request control data structure of type **MMSREQ_PEND** holds information regarding this request. In case of an error, the pointer is set to null and **mms_op_err** is written with the error code.

NOTE: In the request control data structure pointed to by **MMSREQ_PEND**, the **req_info_ptr** member points to a data structure of type **READVARS_REQ_INFO**. This is a special structure used transparently by the virtual machine to keep track of the Read request. See page 2-161 for a detailed description of this structure. In most instances, it will not be necessary to deal with this.

mv_read_variables

Usage: This virtual machine function allows issuing a request with the flexibility of the primitive level read request, but with the ability to interpret the results by the virtual machine. Both the primitive data structure (**READ_REQ_INFO**), and a virtual machine data structure (**MV_READ_REQ_INFO**) are used. This function allows the user to read a list of remotely defined variables of any kind, and place them in local memory using locally defined named type definitions. BE SURE TO SET **m_vmd_select** TO POINT TO THE PROPER VMD STRUCTURE BEFORE CALLING THIS FUNCTION. See **Module 4 — VMD Control** starting on page 2-10 for more information.

Function Prototype:
```
MMSREQ_PEND *mv_read_variables (ST_INT chan,
                                ST_INT num_var,
                                READ_REQ_INFO *prim,
                                MV_READ_REQ_INFO *vminfo);
```

Parameters:

`chan` This is the channel on which request is to be sent.

`num_var` This indicates the number of variables to be read in this Read request. It must be equal to the number of individual structures of type **MV_VARDESC** pointed to by **info**.

`prim` This pointer to a structure of type **READ_REQ_INFO** contains the paired primitive level information specifying the way the data is to be interpreted by the virtual machine. THE ARRAY MUST BE COMMITTED UNTIL THE CONFIRM IS RECEIVED. See page 2-153 for more information on this structure.

`vminfo` This pointer to the beginning of an array of structures of type **MV_READ_REQ_INFO** contains the virtual machine level information specifying which variables are to be read. See page 2-162 for a detailed description of this structure.

NOTE: The **prim** argument is used for generating the request, but then it is disregarded by MMS-EASE. The **vminfo** argument is kept track of by MMS-EASE, and is used when the confirm comes in later. The user also may use the **vminfo** argument later; it is pointed to by the **req_info_ptr** member of the **MMSREQ_PEND** structure corresponding to this request.

Return Value:

`MMSREQ_PEND *` This pointer to the request control data structure of type **MMSREQ_PEND** holds information regarding this request. In case of an error, the pointer is set to null and **mms_op_err** is written with the error code.

. . . continued on the following page . . .

mv_read_variables . . . cont'd from preceding page . . .

NOTE ON ALTERNATE ACCESS:

This **MV_READ_REQ_INFO** structure was not used by the Virtual Machine function so this should not cause any migration problems as long as **mv_read_req_info->i.alt_acc_pres** has been set = **SD_FALSE**.

If the **mv_read_req_info->i.alt_acc_pres** flag is set for a **mv_read_req_info** element and:

1. The primitive read request information specifies a list of variables.
2. The corresponding primitive information 'variable list' structure has the **alt_access_pres** flag set = **SD_FALSE**.

then the Virtual Machine function will create the ASN.1 alternate access and insert it into the variable list element. This ASN.1 Alternate Access will be freed automatically after the primitive request has been issued. No further action is required. Note that you are **ALWAYS** responsible for supplying a valid **mv_read_req_info->i.alt_acc** member, regardless of how the ASN.1 Alternate Access is created.

When the confirm is received, data will be extracted as before, except that if the **mv_read_req_info->i.alt_acc_pres** flag is **SD_TRUE**, the alternate access specified by the **mv_read_req_info->i.alt_acc** element will be used in the conversion to local format. The **mv_read_req_info->i.alt_acc_data_packed** flag will then determine whether the local data is to be packed or not for this element.

NOTE: This means that the **alt_acc** member and its referenced contents must remain committed until the confirm is received.

mv_read_resp

Usage: This virtual machine function allows the user to respond to a Read indication for named variables that have been added to the Virtual Machine NAMED VARIABLE database. Note that this function allows the user to respond to requests for reading named variables. If data is to be read by other means such as by described variables, the primitive function **mp_read_resp** or the VMI function **mv_read_response** must be used. For named variables, **mv_read_resp** accesses the named variable and type definition structures containing the necessary information about the variable. This function will allow multiple named variables to be read in the same read indication. It will make a call to **u_get_named_addr** and **read_ind_fun** for each named variable to be read. See page 2-104 for a detailed description of the sequence of events that occur. BE SURE TO SET **mms_chan_info.objs.vmd** TO POINT TO THE PROPER VMD STRUCTURE BEFORE CALLING THIS FUNCTION. See **Volume 1 — Module 2 — Channel Information Structure** and **Module 4 — VMD Control** starting on page 2-10 for more information.

Function Prototype: `ST_RET mv_read_resp (MMSREQ_IND *ind);`

Parameters:

`ind` This pointer to the indication control structure of type **MMSREQ_IND** is received in the **u_read_ind** function.

Return Value: `ST_RET` **SD_SUCCESS**. No Error

`<> 0` Error Code.

mv_read_response

Usage: This virtual machine function allows the user to respond to Read requests for any type of read indication. It requires that the user sift through all the primitive request information received in the indication (pointed to by **ind->req_info_ptr**), figure out what variables are trying to be read, prepare a **MV_READ_RESP_INFO** structure for each, and then call the **mv_read_response** function. It is up to the user to call any address resolution functions, access the **NAMED_VAR** structures in the Virtual Machine Database in the case of named variables, etc. For each variable that the remote Client is requesting to read, the **mv_read_response** function requires a pointer to a **NAMED_TYPE** structure in the Virtual Machine Database. When the **mv_read_response** function is called, MMS-EASE automatically converts the specified data into the ASN.1 encoding format and sends the Read response. Because **mv_read_response** uses the Virtual Machine Database, BE SURE TO SET **mms_chan_info.objs.vmd** TO POINT TO THE PROPER VMD STRUCTURE BEFORE CALLING THIS FUNCTION. See **Volume 1 — Module 2 — Channel Information Structure** and **Module 4 — VMD Control** starting on page 2-10 for more information.

Function Prototype:

```
ST_RET mv_read_response (MMSREQ_IND *ind,          ST_INT
                         num_var,          MV_READ_RE-
                         SP_INFO *info);
```

Parameters:

`ind` This pointer to the indication control structure of type **MMSREQ_IND** is received in the **u_read_ind** function.

`num_var` This indicates the number of variables read in this Read response. It must be equal to the number of individual structures of type **MV_READ_RESP_INFO** pointed to by **info**. It also should equal the number of variables specified in the request information (**ind->resp_info_ptr->va_spec.num_of_variables**).

`info` This pointer to the beginning of an array of structures of type **MV_READ_RESP_INFO** contains the virtual machine level information specifying which variables are to be responded to. See page 2-162 for a detailed description of this structure.

Return Value: `ST_RET` **SD_SUCCESS**. No Error

`<> 0` Error Code.

NOTE: Although using this function is considered to be more trouble than using the **mv_read_resp** function, it allows for any kind of variable to be responded to, including described variables. It allows for an asynchronous response when necessary. This is unlike the **mv_read_resp** function, which takes over for however long is required to get the data and send the response.

. . . Continued on the following page . . .

mv_read_response . . . cont'd from preceeding page . . .

NOTES ON ALTERNATE ACCESS:

The **MV_READ_RESP_INFO** structure should not cause any migration problems, as long as you have been setting the **mv_read_resp_info->alt_acc_pres** to **SD_FALSE**.

If the **alt_acc_pres** flag is **SD_TRUE**, this function will make use of the **alt_acc** member to convert the local data to ASN.1 for the response.

If:

1. The **mv_read_resp_info->alt_acc_pres** flag is **SD_TRUE**,
2. the **mv_read_resp_info->alt_acc.num_aa** is **0**,
3. this is a list of variables, and
4. the corresponding variable list has an Alternate Access on it,

then the ASN.1 Alternate Access for the variable will be parsed and used.

If:

1. The **mv_read_resp_info->alt_acc_pres** flag is **SD_TRUE**,
2. The **mv_read_resp_info->alt_acc.num_aa** is **!= 0**,

then the user supplied ASN.1 Alternate Access for the variable will be used.

u_mv_read_conf

Usage: This virtual user confirmation function is called when the confirm to a **mv_read** or **mv_readvars** request is received. **resp_err** contains a value indicating whether an error occurred. **resp_info_ptr** may contain data that the remote node sent back in the confirmation or it may contain error information if an error occurred. If no error has occurred (**req->resp_err = CNF_RESP_OK**), the data read is available in the memory location(s) specified in the request information.

See **Volume 1 — Module 1 — Virtual User Confirmation Function Class** for specific information on recommended handling of confirms.

Function Prototype:

```
ST_VOID u_mv_read_conf (MMSREQ_PEND *req);
```

Parameters:

`req` This pointer to the **MMSREQ_PEND** structure is returned from the original **mv_read** or **mv_readvars** request function. See **Volume 1 — Module 2 — Request and Indication Control Data Structures** for more information on this structure.

For a positive response (**req->resp_err = CNF_RESP_OK**), the request control structure member **resp_info_ptr** is a pointer to the operation-specific data structure (**READ_RESP_INFO**) for the Read function. See page 2-153 for more information on this structure. Note that this response information is not normally of interest, since the data is written directly to local memory locations as specified in the original request.

For a negative response, or other errors, see **Volume 3 — Module 11 — MMS-EASE Error Handling** for more information.

Return Value: `ST_VOID` (ignored)

NOTE: Before returning from this function, **ms_clr_mvreq** must be called. This clears up and frees the data used by the virtual machine to handle the request and confirmation.

u_mv_read_vars_conf

Usage: This virtual user confirmation function is called when the confirm to a **mv_read_variables** request is received. **resp_err** contains a value indicating whether an error occurred. **resp_info_ptr** may contain data that the remote node sent back in the confirmation, or it may contain error information if an error occurred. If no error has occurred (**req->resp_err = CNF_RESP_OK**), the data read is available in the memory location(s) specified in the request information. See **Volume 1 — Module 1 — Virtual User Confirmation Function Class** for specific information regarding the recommended handling of confirms.

Function Prototype:

```
ST_VOID u_mv_read_vars_conf (MMSREQ_PEND *req);
```

Parameters:

`req` This pointer to the **MMSREQ_PEND** structure is returned from the original **mv_read_variables** function. See **Volume 1 — Module 2 — Request and Indication Control Data Structures** for more information on this structure.

For a positive response (**req->resp_err = CNF_RESP_OK**), the request control structure member **resp_info_ptr** is a pointer to the operation-specific data structure (**READ_RESP_INFO**) for the Read function. See page 2-154 for more information on this structure. Note that this response information is not normally of interest, since the data is written directly to local memory locations as specified in the original request.

For a negative response, or other errors, see **Volume 3 — Module 11 — MMS-EASE Error Handling** for more information.

Return Value: `ST_VOID` (ignored)

NOTE: Before returning from any virtual user confirmation function, **ms_clr_mvreq** must be called. This clears up and frees the data used by the virtual machine to handle the request and confirmation.

5. Write Service

This service is used for a Client application to request that the Server VMD replace the contents of one or more variables at a remote node with supplied values. MMS-EASE provides both Client and Server functionality.

As mentioned on page 2-142, there are several levels of complexity provided in MMS-EASE for the Write service. The PPI is the most complex and requires the most work by the user application. The easiest request functions to use are **mv_write** and **mv_write_vars**, together with the corresponding confirm function, **u_mv_write_conf**. These functions will write to named variables only. The request function **mv_write_variables** and its confirm function **u_mv_write_vars_conf** provide almost as much power as the PPI (including writing to described variables). Because of this, they are somewhat more complex than **mv_write** and **mv_writevars**. The VMI response function **mv_write_resp** is much easier to use than the PPI. However, it requires a Virtual Machine Database to have been configured ahead of time, and its operation is restricted to named variables only. For the more general case, the support function **ms_extract_write_data** can make the PPI easy to use at the responder. All these functions will be described in this section.

Primitive Level Write Operations

The following section contains information on how to use the paired primitive interface for the Write service. It covers available data structures used by the PPI, and the four primitive level functions that together make up the Write service. See **Volume 1 — Module 2 — General Sequence of Events — Confirmed Services** for a detailed explanation of the sequence of events that occur within the MMS-EASE environment during confirmed services.

- The Write service consists of the paired primitive functions of **mp_write**, **u_write_ind**, **mp_write_resp**, and **u_mp_write_conf**.

Data Structures

Request/Indication

This operation-specific data structure described below is used by the Client in issuing a variable write request (**mp_write**). It is received by the Server when a variable write indication (**u_write_ind**) is received.

```
struct write_req_info
  {
  ST_INT        num_of_data;
  VAR_ACC_DATA  *va_data;
  VAR_ACC_SPEC  va_spec;
/*VARIABLE_LIST var_list [va_spec.num_of_variables];            */
/*VAR_ACC_DATA  var_data_list [num_of_data];                    */
  };
typedef struct write_req_info WRITE_REQ_INFO;
```

Fields:

num_of_data	This indicates the number of structures in the array of structures pointed to by **va_data**.
va_data	This pointer to **var_data_list** is an array of structures of type **VAR_ACC_DATA** containing the data to be written.
va_spec	This structure of type **VAR_ACC_SPEC** contains the variable access specification information.
var_list	This array of structures of type **VARIABLE_LIST** contains the variable specifications for the list of variables to be written.

`var_data_list` This array of structures of type **VAR_ACC_DATA** contains the data to be written into the specified variables.

NOTES:

1. See the description of the variable access data structures starting on page 2-150 for more information on the **VAR_ACC_DATA**, **VARIABLE_LIST** and **VAR_ACC_SPEC** structures.
2. FOR REQUEST ONLY, when allocating a data structure of type **WRITE_REQ_INFO**, enough memory must be allocated to hold the information for the **var_data_list** and **var_list** members of the structure. For example, the following C statement can be used for a list of variables:

```
info = (WRITE_REQ_INFO *) chk_malloc(sizeof (WRITE_REQ_INFO) + (num_of_variables
            * sizeof(VARIABLE_LIST)) + (num_of_data * sizeof(VAR_ACC_DATA)));
```

Response/Confirm

This operation-specific data structure described on the following page is used by the Server in issuing the variable write response (**mp_write_resp**). It is received by the Client when a variable write confirm (**u_mp_write_conf**) is received.

```
struct write_resp_info
  {
  ST_INT num_of_result;
/*WRITE_RESULT wr_result[num_of_result]                    */
  SD_END_STRUCT
  };
typedef struct write_resp_info WRITE_RESP_INFO;
```

Fields:

`num_of_result` This indicates the number of members in **wr_result**. This should match the number of variables included in the Write request.

`wr_result` This array of structures of type **WRITE_RESULT** contains the results of the write (success or failure) for each variable written. The result array members should align with the **var_list** members included in the Write request. The structure is described below.

```
struct write_result
  {
  ST_INT16    resp_tag;
  ST_INT16    failure;
  SD_END_STRUCT
  };
typedef struct write_result WRITE_RESULT;
```

Fields:

`resp_tag` This is a value indicating success or failure of the write operation:

WR_RSLT_FAILURE. Write failed. See **failure** member below.
WR_RSLT_SUCCESS. Write Succeeded.

`failure` This indicates the reason for failure of the write (**resp_tag = WR_RSLT_FAILURE**):

ARE_OBJ_INVALIDATED. An underlying variable of the scattered access or named variable list object was deleted.

ARE_HW_FAULT. An attempt to access the variable has failed due to a hardware fault.

ARE_TEMP_UNAVAIL.The requested variable is temporarily unavailable for the requested access.

ARE_OBJ_ACCESS_DENIED. The MMS Client has insufficient privilege to request this operation.

ARE_OBJ_UNDEFINED. The object with the desired name does not exist.

ARE_INVAL_ADDR. Reference to the unnamed variable object's specified address is invalid because the specified format is incorrect or is out of range.

ARE_TYPE_UNSUPPORTED. An inappropriate or unsupported type is specified for a variable.

ARE_TYPE_INCONSISTENT. A type is specified that is inconsistent with the service or referenced object.

ARE_OBJ_ATTR_INCONSISTENT. The object is specified with inconsistent attributes.

ARE_OBJ_ACC_UNSUPPORTED. The variable is not defined to allow requested access.

ARE_OBJ_NONEXISTENT. The variable is non-existent.

NOTE: FOR RESPONSE ONLY, when allocating a data structure of type **WRITE_RESP_INFO**, enough memory must be allocated to hold the information for the **wr_result** members of the structure. For example, the following C statement can be used:

```
info = (WRITE_RESP_INFO *) chk_malloc (sizeof (WRITE_RESP_INFO) + num_of_result *
            sizeof (WRITE_RESULT));
```

Paired Primitive Interface Functions

mp_write

Usage: This primitive request function sends a Write request PDU using the data from a structure of type **WRITE_REQ_INFO**, pointed to by **info**. It is used to write a value into a variable or list of variables that exist at the remote node.

Function Prototype:

```
MMSREQ_PEND *mp_write (ST_INT chan,
                       WRITE_REQ_INFO *info);
```

Parameters:

chan	This integer contains the channel number over which the Write PDU should be sent.
info	This pointer to Operation-Specific data structure of type **WRITE_REQ_INFO** contains information specific to the Write PDU to be sent.

Return Value:

MMSREQ_PEND *	This pointer to the request control data structure of type **MMSREQ_PEND** is used to send the write PDU. In case of an error, the pointer is set to null and **mms_op_err** is written with the error code.

Corresponding User Confirmation Function: **u_mp_write_conf**

Operation-Specific Data Structure Used: **WRITE_REQ_INFO**

See page 2-173 for more a detailed description of this structure.

u_write_ind

Usage: This user function is called when a Write indication is received by a Server node. The contents and operation of this function are user-defined, and depend on the application requirements. To conform to the MMS protocol requirements, the application must do one of the following some time after the indication is received:

1) send a positive response using either:

   a) the primitive response function (**`mp_write_resp`**) after writing the specified variable(s),

   b) the virtual machine response function (**`mv_write_resp`**) to have the virtual machine take the requested action and send the response, or

2) call the appropriate primitive negative (error) response function (**`mp_err_resp`**). See **Volume 3 — Module 11 — MMS-EASE Error Handling** for an explanation of this function.

See **Volume 1 — Module 1 — User Indication Function Class** for additional information regarding the handling of indications.

Function Prototype: `ST_VOID u_write_ind (MMSREQ_IND *ind);`

Parameters:

`ind` This pointer to the **`MMSREQ_IND`** structure is described in **Volume 1 — Module 2 — Request and Indication Control Data Structures**. This pointer must be used to send the response to this indication. The primary item of interest in this structure is **`ind->req_info_ptr`**. This is a pointer to the operation-specific data structure for a Write indication (**`WRITE_REQ_INFO`**). This pointer will always be valid when **`u_write_ind`** is called.

Return Value: `ST_VOID` (ignored)

Operation-Specific Data Structure Used: **`WRITE_REQ_INFO`**

See page 2-173 for a detailed description of this structure.

mp_write_resp

Usage: This primitive response function sends a Write positive response PDU. This function should be called after the **u_write_ind** function is called (a Write indication is received), and after the variable(s) have been successfully written.

Function Prototype:

```
ST_RET mp_write_resp (MMSREQ_IND *ind,
                      WRITE_RESP_INFO *info);
```

Parameters:

`ind` This pointer to the indication control structure of type **MMSREQ_IND** is received in the service function, **u_write_ind**.

`info` This pointer to an Operation-Specific data structure of type **WRITE_RESP_INFO** contains information specific to the PDU to be sent.

Return Value: `ST_RET` **SD_SUCCESS**. No Error.

`<> 0` Error Code.

Corresponding User Indication Function: **u_write_ind**

Operation-Specific Data Structure Used: **WRITE_RESP_INFO**

See page 2-174 for a detailed description of this structure.

u_mp_write_conf

Usage: This primitive user confirmation function is called when the confirm to a **mp_write** request is received. **resp_err** contains a value indicating whether an error occurred. **resp_info_ptr** may contain data that the remote node sent back in the confirmation or it may contain error information if an error occurred. If no error has occurred (**req->resp_err = CNF_RESP_OK**), write information is available to the application using the **req->resp_info_ptr** pointer. See **Volume 1 — Module 1 — Virtual User Confirmation Function Class** for specific information regarding the recommended handling of confirms.

Note that the function called for this event may be changed if desired; however, **u_mp_write_conf** is the standard default function.

Function Prototype:

```
ST_VOID u_mp_write_conf (MMSREQ_PEND *req);
```

Parameters:

`req` This pointer to the **MMSREQ_PEND** structure is returned from the original **mp_write** request function. See **Volume 1 — Module 2 — Request and Indication Control Data Structures** for more information on this structure.

For a positive response (**req->resp_err = CNF_RESP_OK**), the request control structure member **resp_info_ptr** is a pointer to the operation-specific data structure (**WRITE_RESP_INFO**) for the Write response.

For a negative response, or other errors, see **Volume 3 — Module 11 — MMS-EASE Error Handling** for more information.

Return Value: `ST_VOID` (ignored)

Operation-Specific Data Structure Used: **WRITE_RESP_INFO**

See page 2-174 for a detailed description of this structure.

Virtual Machine Write Operations

The following section contains information on how to use the virtual machine interface for the Write service. It covers available data structures used by the VMI, and the functions that together make up the Write service.

- The Write service also consists of the virtual machine functions of **mv_write**, **mv_writevars**, **mv_write_variables**, **mv_write_resp**, **u_mv_write_conf**, and **u_mv_write_vars_conf**. Note that the primitive indication function **u_write_ind** is also used with the VMI. SISCO recommends using the virtual machine function for the Write service. This portion of the Write service for the virtual machine has some exceptions to the standard sequence for virtual machine:

1. Node A sends a named variable write request by calling one of the following:

   mv_write allows the ability to write one named variable at the remote node based on a locally defined named type definition.

   mv_writevars allows writing a **LIST** of remotely defined named variables at the remote node based on locally defined named type definitions.

   mv_write_variables allows writing a **LIST** of remotely defined variables (named and unnamed) at the remote node based on locally defined named type definitions. This function allows issuing a request with the flexibility of a primitive level request but without having to use the ASN.1 <-> local conversion utility.

   NOTE: SISCO recommends the use of the **mv_write_variables** function over the PPI. This function allows writing named, addressed, and described variables. When only named variables are involved, SISCO recommends the use of **mv_write** and **mv_writevars**.

2. Within the user-defined **u_write_ind** function on Node B, the user can call the following VMI response function: **mv_write_resp**. Alternatively, the user could use the support function **ms_extract_write_data** with the primitive response function **mp_write_resp**.

   This rest of this sequence deals with calling **mv_write_resp**.

3. The MMS-EASE on Node B sorts through the operation-specific data corresponding to the write indication received and finds the variable and type information corresponding to the variable being written. MMS-EASE then calls the **u_get_named_addr** function and passes the address information found in its database to this function.

4. The user's program on Node B determines the variable's physical address of the variable, and returns that address from **u_get_named_addr**.

5. MMS-EASE on Node B calls the **write_ind_fun** pointed to in the **NAMED_VAR** structure for the variable being written. If the user has modified the **write_ind_fun** from the default, the physical address of the variable must be returned. If the user has not changed the default **write_ind_fun**, no further action is necessary at this step.

6. MMS-EASE on Node B converts the data in the Write request to the local C language format. It stores the data at the address returned from **write_ind_fun** using the type definition information found in the virtual machine's database for the variable being written. It then generates this data into a positive write response PDU. The PDU is sent and a value of SUCCESS is returned to the user's program on Node B that called **mv_write_resp**.

7. If at any time during this processing by the virtual machine, an error is detected, or if a null pointer is returned from either **u_get_named_addr** or **write_ind_fun**, an error response is sent back to Node A. Then, an error code is returned to the user's program on Node B that called **mv_write_resp**.

8. At Node A, MMS-EASE receives the confirm and calls **u_mv_write_conf**.

Data Structures

Request/Indication

The following data structure is used by the Virtual Machine **mv_write_variables** request function. See page 2-184 for more information on this function. An array of these structures is used with this function, one per variable to be written.

```
struct mv_write_req_info
  {
  ST_CHAR    *data_ptr;
  NAMED_TYPE *type;
  ST_BOOLEAN alt_acc_pres;
  ST_BOOLEAN alt_acc_data_packed;
  ALT_ACCESS alt_acc;
  };
typedef struct mv_write_req_info MV_WRITE_REQ_INFO;
```

Fields:

`data_ptr` This is the pointer to the local data to be written to the corresponding remote variable.

`type` This structure of type **NAMED_TYPE** contains the type associated with this variable. See page 2-62 for more information on this data structure.

`alt_acc_data_packed`

SD_TRUE. The data present in **alt_acc** is packed.

SD_FALSE. The data present in **alt_acc** is not packed.

`alt_acc` This structure of type **ALT_ACCESS** contains the alternate access description. See page 2-0 for more information on this structure.

Virtual Machine Interface Functions

mv_write

Usage: This virtual machine function allows the user to execute the Write service without having to deal directly with the operation-specific data structures. This function allows the user to write a named variable at the remote application based upon a locally defined named type definition. BE SURE TO SET **`m_vmd_select`** TO POINT TO THE PROPER VMD STRUCTURE BEFORE CALLING THIS FUNCTION. See **Module 4 — VMD Control** starting on page 2-10 for more information.

Function Prototype:

```
MMSREQ_PEND *mv_write (ST_INT chan,
                       OBJECT_NAME *var,
                       OBJECT_NAME *type,
                       ST_CHAR *src);
```

Parameters:

`chan` This is the channel on which request is to be sent.

`var` This pointer to a structure of type **`OBJECT_NAME`** contains the name of the remotely defined named variable that is to be written to. See **Volume 1 — Module 2 — MMS Object Name** for more information on this structure.

`type` This pointer to a structure of **`OBJECT_NAME`** contains the name of a locally defined named type concerning to the variable described by **`var`**.

`src` This is a pointer to where the data can be found that should be written into the remote variable.

Return Value:

`MMSREQ_PEND *` This pointer to a request control data structure of type **`MMREQ_PEND`** contains information regarding this request. In case of an error, the pointer is set to null and **`mms_op_err`** is written with the error code.

mv_writevars

Usage: This virtual machine function allows the user to execute the Write service without having to deal directly with the operation-specific data structures. This function allows the user to write a list of remotely defined named variables at the remote application based upon locally-defined named type definition. BE SURE TO SET **`m_vmd_select`** TO POINT TO THE PROPER VMD STRUCTURE BEFORE CALLING THIS FUNCTION. See **Module 4 —VMD Control** starting on page 2-10 for more information.

Function Prototype:

```
MMSREQ_PEND *mv_writevars (ST_INT chan,
                           ST_INT num_var,
                           MV_VARDESC *info);
```

Parameters:

`chan` This is the channel on which request is to be sent.

`num_var` This indicates the number of variables to be written in this Write request. It must be equal to the number of individual structures of type **`MV_VARDESC`** that are pointed to by **`info`**.

`info` This pointer to the beginning of an array of structures of type **`MV_VARDESC`** contains the information specifying the variables to be written. See page 2-143 for more information on this structure.

Return Value:

`MMSREQ_PEND *` This pointer to the request control data structure of type **`MMSREQ_PEND`** contains information regarding this request. In case of an error, the pointer is set to null and **`mms_op_err`** is written with the error code.

mv_write_variables

Usage: This virtual machine function allows the user to issue a request with the flexibility of the primitive level write request, but without having to use the ASN.1 <-> local conversion utilities. The virtual machine data structure (**MV_WRITE_REQ_INFO**) is used for inputting information. This allows writing a list of remotely defined named variables of any kind at the remote node based upon locally defined named type definitions. BE SURE TO SET **m_vmd_select** TO POINT TO THE PROPER VMD STRUCTURE BEFORE CALLING THIS FUNCTION. See **Module 4 — VMD Control** starting on page 2-10 for more information.

Function Prototype:

```
MMSREQ_PEND *mv_write_variables (ST_INT chan,
                                 ST_INT num_data,
                                 VAR_ACC_SPEC *reqinfo,
                                 MV_WRITE_REQ_INFO *vminfo);
```

Parameters:

`chan` This is the channel on which request is to be sent.

`num_data` This indicates the number of structures in the array of structures pointed to by **vminfo**.

`reqinfo` This pointer to ~~an array~~ of structures of type **VAR_ACC_SPEC** contains the information specifying the variables to be written. See page 2-150 for more information on this structure.

`vminfo` This pointer to an array of structures of type **MV_WRITE_REQ_INFO** contains information representing the data for each variable to be encoded in the request. This information can be freed immediately after the call to this function. See page 2-181 for information on this structure.

Return Value:

`MMSREQ_PEND *` This pointer to the request control data structure of type **MMSREQ_PEND** contains information regarding this request. However, no request information is kept track of with this function, and the **req_info_ptr** member of **MMSREQ_PEND** structure corresponding to this request, is not used. In case of an error, the pointer is set to null and **mms_op_err** is written with the error code.

. . . continued on the following page . . .

mv_write_variables . . . cont'd from preceding page . . .

NOTES ON ALTERNATE ACCESS:

The **alt_acc_pres** flag now must be set **SD_FALSE** to maintain compatibility with the old version of this function. If the **alt_acc_pres** flag is set for a **MV_WRITE_REQ_INFO** member and :

1. the primitive read request information specifies a list of variables
2. the corresponding primitive information **variable_list** structure has the **alt_access_pres flag = SD_FALSE**

then the Virtual Machine function will create the ASN.1 alternate access and insert it into the variable list element. This will be freed automatically after the primitive request has been issued, and no further action is required. Note that you are ALWAYS responsible for supplying a valid **alt_acc** member, regardless of how the ASN.1 Alternate Access is created. In creating the ASN.1 data elements, data will be encoded as before, except that if the **mv_write_req_info ->alt_acc_pres** flag is **SD_TRUE**, the alternate access specified by the **alt_acc** element will be used in the conversion from local format to ASN.1. The **mv_write_req_info->alt_acc_data_packed** flag will then determine whether the local data is assumed to be packed or not for this element. After the return from this function, all information may be freed if desired; no commitment is required.

mv_write_resp

Usage: This virtual machine function allows the user to respond to a Write indication without actually having to directly manipulate the data that is being written. This function allows the incoming data to be written to named variables only. Otherwise, the user must handle the data directly, and then call the primitive function **mp_write_resp** to send the response. For named variables, **mv_write_resp** automatically accesses the appropriate named variable information in the Virtual Machine Database. This function will allow multiple named variables to be written in the same write indication. It will make a single call to **u_get_named_addr** and **write_ind_fun** for each named variable to be written. BE SURE TO SET **mms_chan_info.objs.vmd** TO POINT TO THE PROPER VMD STRUCTURE BEFORE CALLING THIS FUNCTION. See **Volume 1 — Module 2 — Channel Information Structure** and **Module 4 — VMD Control** starting on page 2-10 for more information.

Function Prototype: `ST_RET mv_write_resp (MMSREQ_IND *ind);`

Parameters:

`ind` This pointer to the request data structure of type **MMSREQ_IND** is received in the **u_write_ind** function.

Return Value: `ST_RET` **SD_SUCCESS**. No Error.

`<> 0` Error Code.

u_mv_write_conf

Usage: This virtual user confirmation function is called when a confirm to a **mv_write** or **mv_writevars** request is received. **resp_err** variable contains a value indicating whether an error occurred. **resp_info_ptr** may contain data that the remote node sent back in the confirmation or it may contain error information if an error occurred. If no error has occurred (**req->resp_err = CNF_RESP_OK**), Write information is available to the application using the **req->resp_info_ptr** pointer. This information is just the status (success or failure) of the Write operation for each variable.

See **Volume 1 — Module 1 — Virtual User Confirmation Function Class** for specific information on the recommended handling of confirms.

Function Prototype:

```
ST_VOID u_mv_write_conf (MMSREQ_PEND *req);
```

Parameters:

`req` This pointer to the **MMSREQ_PEND** structure is returned from the original **mv_write** or **mv_writevars** request function. See **Volume 1 — Module 2 — Request and Indication Control Data Structures** for more information on this structure.

For a positive response (**req->resp_err = CNF_RESP_OK**), the request control structure member **resp_info_ptr** is a pointer to the operation-specific data structure (**WRITE_RESP_INFO**) for the Write function.

For a negative response, or other errors, see **Volume 3 — Module 11 — MMS-EASE Error Handling** for more information.

Return Value: ST_VOID (ignored)

Operation-Specific Data Structure Used: **WRITE_RESP_INFO**

See page 2-174 for a detailed description of this structure.

NOTE: Before returning from any virtual user confirmation function, **ms_clr_mvreq** must be called. It clears up and frees the data used by the virtual machine to handle the request and confirmation.

u_mv_write_vars_conf

Usage: This virtual user confirmation function is called when the confirm to a **mv_write_variables** request is received. **resp_err** variable contains a value indicating whether an error occurred. **resp_info_ptr** may contain data that the remote node sent back in the confirmation or it may contain error information, if an error occurred. If no error has occurred (**req->resp_err = CNF_RESP_OK**), Write information is available to the application using the **req->resp_info_ptr** pointer. This information is just the status (success or failure) of the Write operation for each variable.

See **Volume 1 — Module 1 — Virtual User Confirmation Function Class** for specific information on recommended handling of confirms.

Function Prototype:

```
ST_VOID u_mv_write_vars_conf (MMSREQ_PEND *req);
```

Parameters:

`req` This pointer to the **MMSREQ_PEND** structure is returned from the original **mv_write_variables** request function. See **Volume 1 — Module 2 — Request and Indication Control Data Structure** for more information on this structure.

For a positive response (**req->resp_err = CNF_RESP_OK**), the request control structure member **resp_info_ptr** is a pointer to the operation-specific data structure (**WRITE_RESP_INFO**) for the Write function.

For a negative response, or other errors, see **Volume 3 — Module 11 — MMS-EASE Error Handling** for more information.

Return Value: `ST_VOID` (ignored)

Operation-Specific Data Structure Used: **WRITE_RESP_INFO**

See page 2-174 for a detailed description of this structure

NOTE: Before returning from any virtual user confirmation function, **ms_clr_mvreq** must be called. It clears up and frees the data used by the virtual machine to handle the request and confirmation.

ms_extract_write_data

Usage: This support function extracts the data corresponding to the **i**th variable (where **i** is the index) from a Write PDU. It converts it from the ASN.1 data representation (using a named type already existing in the MMS-EASE database), and puts the data into the user-specified memory address. Any AlternateAccess present for the **i**th variable is converted from its ASN.1 form and applied to the derived type. Use this function, within the **u_write_ind** function, to extract the variable information contained in a Write indication. See the **ms_extract_varname** function for information on determining the names of the variables included in a Write PDU when named variables are used.

Function Prototype:

```
ST_RET ms_extract_write_data (MMSREQ_IND *ind,
                              ST_INT index,
                              NAMED_TYPE *type,
                              ST_CHAR *dest,
                              ST_BOOLEAN alt_acc_packed);
```

Parameters:

`ind` This pointer to the indication control structure of type **MMSREQ_IND** is passed to the **u_write_ind** function when it was called.

`index` This is an index indicating the variable, out of the list of variables contained in the Write PDU, for which the data is to be extracted and converted. If this parameter equals 9, then the 10th variable is extracted and converted. The first variable in the list of variables would have an index of zero (0).

`type` This pointer to the **NAMED_TYPE** structure contains the named type definition existing in the MMS-EASE database. Use **ms_find_named_type_obj** to obtain this pointer.

`dest` This is a pointer to where the converted data is to be put. This should point to a 'C' variable or structure corresponding to the type definition for the variable being extracted.

`alt_acc_packed` This flag indicates whether the data is to be packed when alternate access is used. The default value is **SD_FALSE**, not packed.

Return Value: `ST_RET` **SD_SUCCESS**. No error.

`<> 0` Error Code.

NOTES ON ALTERNATE ACCESS:

This function now will handle alternate access when the indication is a list of variables. In this case, the Virtual Machine functions will use the received ASN.1 alternate access in the conversion to local format.

The **alt_acc_data_packed** parameter will then determine whether the local data is assumed to be packed or not for this element. In prior versions of this function, this parameter was ignored.

NOTE: This does not provide a good solution for a VariableList write where the VariableList is made up of variables with associated alternate access. For this situation, use the lower level data conversion functions to get the desired results.

ms_extract_wr_data

Usage: This support function extracts the data corresponding to the **i**th variable (where **i** is the index) from a Write PDU. It converts it from the ASN.1 data representation (using a named type already existing in the MMS-EASE database), and puts the data into the user-specified memory address. Any AlternateAccess present for the **i**th variable is converted from its ASN.1 form and applied to the derived type. The application will be returned the decoded AlternateAccess when the **alt_acc_ptr** is not passed in as a null. Use this function, within the **u_write_ind** function, to extract the variable information contained in a Write indication. See the **ms_extract_varname** function for information on determining the names of the variables included in a Write PDU when named variables are used.

Function Prototype:

```
ST_RET ms_extract_wr_data (MMSREQ_IND *indptr,
                           ST_INT i,
                           NAMED_TYPE *type,
                           ST_CHAR *dest,
                           ST_BOOLEAN alt_acc_packed,
                           ALT_ACCESS *alt_acc_ptr);
```

Parameters:

`indptr` This pointer to the indication control structure of type **MMSREQ_IND** is passed to the **u_write_ind** function when it was called.

`i` This is an index indicating the variable, out of the list of variables contained in the Write PDU, for which the data is to be extracted and converted. If this parameter equals 9, then the 10th variable is extracted and converted. The first variable in the list of variables would have an index of zero (0).

`type` This pointer to the **NAMED_TYPE** structure contains the named type definition existing in the MMS-EASE database. Use **ms_find_named_type_obj** to obtain this pointer.

`dest` This is a pointer to where the converted data is to be put. This should point to a 'C' variable or structure corresponding to the type definition for the variable being extracted.

`alt_acc_packed` This flag indicates whether the data is to be packed when alternate access is used. The default value is **SD_FALSE**, not packed.

`alt_acc_ptr` This is an input and output parameter. The function will always decode and apply AlternateAccess to the derived type.

As an input parameter it tells the function if it should return the decoded AlternateAccess to the application. If the **alt_acc_ptr** is supplied as a null to the function no AlternateAccess information will be returned.

As an output parameter, this is the address of an **ALT_ACCESS** structure that contains the decoded alternate access definition when alternate access is present in the Write service request. When AlternateAccess information is returned it is the responsibility of the application to free the dynamic memory involved.

. . . continued on the following page . . .

ms_extract_wr_data . . . cont'd from preceding page . . .

Return Value: ST_RET **SD_SUCCESS**. No error.

<> 0 Error Code.

NOTES ON ALTERNATE ACCESS:

This function now will handle alternate access when the indication is a list of variables. In this case, the Virtual Machine functions will use the received ASN.1 alternate access in the conversion to local format.

The **alt_acc_data_packed** parameter will then determine whether the local data is assumed to be packed or not for this element. In prior versions of this function, this parameter was ignored.

This does not provide a good solution for a VariableList write where the VariableList is made up of variables with associated alternate access. For this situation, use the lower level data conversion functions to get the desired results.

mv_write_decode

Usage: This function has been added to MMS-EASE for the very special case when the user may want the Virtual Machine to decode a Write Indication without automatically sending the positive response. This allows the user to perform additional validation on the data before sending a response. This function does everything that **mv_write_resp** does except sending the response. This includes decoding the ASN.1 representation of the variable data and writing it into local memory at the address defined for the variable. It allocates the **WRITE_RESP_INFO** structure and returns a pointer to it. It is the responsibility of the user to call **mp_write_resp** (or **mp_err_resp**) after calling this function and then free this structure using **chk_free**.

Function Prototype:

```
WRITE_RESP_INFO *mv_write_decode (MMSREQ_IND *indptr);
```

Parameters:

`indptr` This pointer to the indication control structure of type **MMSREQ_IND** points to the Indication Information structure passed to **u_getvlist_ind**.

Return Value:

`MMSREQ_PEND *` This pointer to the request control data structure of type **MMSREQ_PEND** is used in a **mp_write_resp** call (BE SURE TO FREE THIS WITH **chk_free**). It is set to null if write data could not be decoded (**mp_err_resp** has already been called and **mms_op_err** is written with the error code).

6. InformationReport Service

This service is used to inform the other node of the value of one or more specified variables, as read by the issuing node.

As mentioned on page 2-85, there are several levels of complexity provided in MMS-EASE for the InformationReport service. The PPI is the most complex and requires the most work for the user application. The easiest request functions to use are **mv_info** and **mv_infovars**. These will report information regarding named variables only. The request function **mv_info_report** provides almost as much power as the PPI (including reporting the values of described variables). So, it is somewhat more complex than **mv_info** and **mv_info_vars**. There is also an indication support function, **ms_extract_info_data**, that helps to use the data received in an InformationReport indication. Since this is an unconfirmed service, there are no **u_mv_xxxx_conf** functions. All the functions concerning this service will be described in this section.

Primitive Level InformationReport Operations

The following section contains information on how to use the paired primitive interface for the InformationReport service. The InformationReport service is an unconfirmed service. This section covers available data structures used by the PPI, and the two primitive level functions that together make up the InformationReport service. See **Volume 1 — Module 2 — General Sequence of Events — Unconfirmed Services** for a detailed explanation of the sequence of events that occur within the MMS-EASE environment during unconfirmed services.

- The InformationReport service consists of the paired primitive functions of **mp_info**, and **u_info_ind**.

Data Structures

Request/Indication

The operation-specific data structure described below is used by the Client in issuing the request (**mp_info**). It is received by the Server when an InformationReport indication (**u_info_ind**) is received.

```
struct info_req_info
  {
  ST_INT         num_of_acc_result;
  ACCESS_RESULT *acc_rslt_list;
  VAR_ACC_SPEC  va_spec;
/*VARIABLE_LIST var_list [va_spec.num_of_variables];             */
/*ACCESS_RESULT acc_result_list [num_of_acc_result];             */
/*SD_END_STRUCT                                                      */
  };
typedef struct info_req_info INFO_REQ_INFO;
```

Fields:

`num_of_acc_result` This indicates the number of access results in the array of structures pointed to by **acc_rslt_list**.

`acc_rslt_list` This is a pointer to **acc_result_list**. This array of structures of type **ACCESS_RESULT** contains the values of the variables being reported on in the InformationReport.

`va_spec` This structure of type **VAR_ACC_SPEC** specifies on which variables are being reported.

`var_list` This array of structures of type **VARIABLE_LIST** contains the list of variables included in this PDU.

`acc_result_list` This array of structures of type **`ACCESS_RESULT`** contains the access results for the specified variables.

NOTES:

1. See the description of the variable access data structures starting on page 2-150 for more information on the **`ACCESS_RESULT`**, **`VARIABLE_LIST`** and **`VAR_ACC_SPEC`** data structures.
2. FOR REQUEST ONLY, when allocating a data structure of type **`INFO_REQ_INFO`**, enough memory must be allocated to hold the information for the **`var_list`** and **`acc_result_list`** members of the structure. The following C statement can be used:

```
info = (INFO_REQ_INFO *) chk_malloc (sizeof (INFO_REQ_INFO) + (num_of_variables *
            sizeof(VARIABLE_LIST)) + (num_of_acc_result *
            sizeof(ACCESS_RESULT)));
```

Paired Primitive Interface Functions

mp_info

Usage: This primitive request function sends an InformationReport PDU using the data from a structure of type **INFO_REQ_INFO**, pointed to by **info**. An InformationReport is used to send, in an unsolicited manner, the contents of a list of local variables to a remote node.

Function Prototype:

```
ST_RET mp_info (ST_INT chan, INFO_REQ_INFO *info);
```

Parameters:

chan	This contains the channel number over which the InformationReport PDU should be sent.
info	This pointer to an Operation-Specific data structure of type **INFO_REQ_INFO** contains information specific to the PDU to be sent.

Return Value:

ST_RET	**SD_SUCCESS**.	The InformationReport was successful.
	<> 0	The InformationReport was not successful. The value of the return code is a MMS-EASE error code. See Appendix A for a description of the MMS-EASE error codes.

Corresponding User Confirmation Function: NONE

Operation-Specific Data Structure Used: **INFO_REQ_INFO**

See page 2-193 for a detailed description of this structure.

u_info_ind

Usage: This user function is called when an InformationReport indication is received by a Server node. The contents and operation of this function are user-defined, and depend on the application requirements.

See **Volume 1 — Module 1 — User Indication Function Class** for additional information on handling indication functions.

Function Prototype: `ST_VOID u_info_ind (MMSREQ_IND *ind);`

Parameters:

`ind` This pointer to the **`MMSREQ_IND`** structure is described in **Volume 1 — Module 2 — Request and Indication Control Data Structures**. This pointer must be used to send the response to this indication. The primary item of interest in this structure is **`req->req_info_ptr`**. This is a pointer to the operation-specific data structure for an Information Report indication (**`INFO_REQ_INFO`**). This pointer will always be valid when **`u_info_ind`** is called.

Return Value: `ST_VOID` (ignored)

Operation-Specific Data Structure Used: **`INFO_REQ_INFO`**

See page 2-193 for a detailed description of this structure.

Virtual Machine InformationReport Operations

The following section contains information on how to use the virtual machine interface for the InformationReport service. It covers available data structures used by the VMI, and the functions that together make up the Information Report service.

- The Information Report service consists of the virtual machine functions of **mv_info**, **mv_infovars**, **mv_info_report**, and **mv_info_nvlist**. The Information Report service, as implemented in the VMI, differs slightly from the standard sequence for unconfirmed services:

1. Node A sends an Information Report request by calling one of the following:

   mv_info allows the ability to report the value of a single variable on Node A to a remote application in an unsolicited manner.

   mv_infovars allows the ability to report the values of a **LIST** of named variables on Node A, in an unsolicited manner.

   mv_info_report allows the ability to report the values of a **LIST** of variables of any kind on Node A, in an unsolicited manner. This function allows issuing a request with the flexibility of a primitive level request, without having to deal with converting data from local representation to ASN.1-encoded format.

   mv_info_nvlist allows the ability to report the values of a NamedVariableList only of any kind on Node A, in an unsolicited manner. This function allows issuing a request with the flexibility of a primitive level request, without having to deal with converting data from local representation to ASN.1-encoded format.

   NOTE: SISCO recommends that the **mv_info_report** function be used, rather than the PPI. If only named variables are involved, we recommend using **mv_info** or **mv_infovars**.

2. Node B receives the Information Report indication, and MMS-EASE calls **u_info_ind**. The user application on Node B can use the indication support functions **ms_extract_info_data** and **ms_extract_varname** to help interpret the information received.

Data Structures

Request/Indication

The following data structure is used by the Virtual Machine **mv_info_report** request function. An array of these structures is used with this function, one structure per variable on which to report. See page 2-201 for more information on this function.

```
struct mv_info_req_info
  {
  ST_CHAR    *data_ptr;
  NAMED_TYPE *type;
  ST_BOOLEAN alt_acc_pres;
  ST_BOOLEAN alt_acc_data_packed;
  ALT_ACCESS alt_acc;
  };
typedef struct mv_info_req_info  MV_INFO_REQ_INFO;
```

Fields:

data_ptr	This pointer to the local format data results from a successful information report request.
type	This structure of type **NAMED_TYPE** contains the type associated with this variable. See page 2-62 for more information on this data structure.

`alt_acc_pres` **`SD_TRUE`**. **`alt_acc`** is present. Alternate Access is used.

`SD_FALSE`. **`alt_acc`** is not present. Alternate Access is not used.

`alt_acc_data_packed` **`SD_TRUE`**. The data present in **`alt_acc`** is packed.

`SD_FALSE`. The data present in **`alt_acc`** is not packed.

`alt_acc` This structure of type **`ALT_ACCESS`** contains the alternate access description. See page 2-0 for more information on this structure.

Virtual Machine Interface Functions

mv_info

Usage: This virtual machine function allows sending an InformationReport PDU. This reports the status of a single named variable to a remote application in an unsolicited manner. It is like sending an unsolicited read response. The variable name is always specified by pointing to it using **var**. If you want to use this function to report an "artificial" variable (one that does not exist in the virtual machine database), the type name (which must be defined locally) must be specified, and passed a pointer to it in **type**. If the variable to be reported does exist (the definition for it is in the Virtual Machine Database), simply leave the pointer to the type name (***type**) set to null. Make sure that **var** points to the name of a defined named variable. In this case, MMS-EASE uses the virtual machine database to obtain the type information. Similarly, if the data pointer argument (***src**) is left null, the virtual machine will use its database to find the data. A corresponding call to **u_get_named_addr** is made to resolve the variable's address. BE SURE TO SET **m_vmd_select** TO POINT TO THE PROPER VMD STRUCTURE BEFORE CALLING THIS FUNCTION. See **Module 4 — VMD Control** starting on page 2-10 for more information.

Function Prototype:

```
ST_RET mv_info (ST_INT chan,
                OBJECT_NAME *var,
                OBJECT_NAME *type,
                ST_CHAR *src);
```

Parameters:

`chan` This contains the number of the channel on which request is to be sent.

`var` This pointer to structure of type **OBJECT_NAME** contains the name you want to be used for the variable whose data is included in the Information Report PDU.

`type` This pointer to structure of type **OBJECT_NAME** contains the name of a locally defined named type pertaining to the variable described by **var**. If this argument is left null, MMS-EASE uses the type information found in the virtual machine database for the variable name specified by **var**.

`src` This is a pointer to where the data included in the InformationReport PDU is to be found. MMS-EASE will use the type definition (pointed by **type** or found in the virtual machine database) to determine the format and length of this data. If this argument is left null, MMS-EASE will use the virtual machine database, and make a call to **u_get_named_addr**. This resolves the physical address of the variable to be reported.

Return Value: ST_RET

SD_SUCCESS. No Error.

<> 0 Error. (Information Report not sent)

mv_infovars

Usage: This virtual machine function allows the user to send an InformationReport PDU without having to deal with the operation-specific data structures directly. This function is used to report the status of a list of named variables to a remote node in an unsolicited manner. It is like sending an unsolicited read response. This function is more general purpose than **mv_info**. Since it can send an InformationReport for any number of variables from 1 to the maximum that can be fit in a single PDU. As in **mv_info**, this function can be used to send Information Reports for "artificial" or defined named variables. To send an Information Report for defined named variables, you would set the **type** and **data** elements equal to null. Elements of one of the structures of type **mv_vardesc** pointed to by **vminfo**. In this case, MMS-EASE will use the virtual machine database to find the type information, and will call **u_get_named_addr** to find the physical address of the variable(s). If you make the **type** and **data** elements of each **MV_VARDESC** structure point to an actual defined named type, and where to get the data for the InformationReport from (respectively), this information is used instead of the virtual machine database. Defined named variables and "artificial" variables can be mixed and matched in the same InformationReport PDU using this function. BE SURE TO SET **m_vmd_select** TO POINT TO THE PROPER VMD STRUCTURE BEFORE CALLING THIS FUNCTION. See **Module 4 — VMD Control** starting on page 2-10 for more information.

Function Prototype:

```
ST_RET mv_infovars (ST_INT chan,
                    ST_INT num_vars,
                    MV_VARDESC *vminfo);
```

Parameters:

chan	The contains the channel number over which the InformationReport PDU is to be sent.
num_vars	This indicates the number of variables to be included in this PDU. It must be equal to the number of individual structures of type **MV_VARDESC** pointed to by **var_desc**.
vminfo	This pointer to the beginning of an array of structures of type **MV_VARDESC** contains the information specifying which variables to be included in this InformationReport PDU. See page 2-143 for more information on this structure.

Return Value: ST_RET **SD_SUCCESS**. No Error.

<> 0 Error. (InformationReport not sent)

mv_info_report

Usage: This virtual machine function allows the user to report the values of a list of variables of any kind to a remote node in an unsolicited manner. It is more general purpose than **mv_infovars**. Since it allows the user the flexibility of the primitive level information report, but with the ability to interpret results as with the virtual machine. It uses the virtual machine data structure **MV_INFO_REQ_INFO**. BE SURE TO SET **m_vmd_select** TO POINT TO THE PROPER VMD STRUCTURE BEFORE CALLING THIS FUNCTION. See **Module 4 — VMD Control** starting on page 2-10 for more information.

Function Prototype:

```
ST_RET mv_info_report (ST_INT chan,
                       VAR_ACC_SPEC *req,
                       ST_INT num_var,
                       MV_INFO_REQ_INFO *vminfo);
```

Parameters:

`chan` This contains the channel number over which the Information Report PDU is to be sent.

`req` This pointer to ~~an array~~ of structures of type **VAR_ACC_SPEC** contains the information specifying which variables are to be included in this Information Report PDU. These variables can be specified by name, or address and type. See page 2-150 for a detailed description of this structure.

`num_var` This indicates the number of variables to be included in this PDU. It must be equal to the number of individual structures of type **MV_INFO_REQ_INFO** pointed to by **vminfo**.

`vminfo` This pointer to an array of structures of type **MV_INFO_REQ_INFO** contains information representing the data for each variable to be encoded in the request. See page 2-197 for a detailed description of this structure.

Return Value: `ST_RET`

SD_SUCCESS. No Error.

`<> 0` Error. (InformationReport not sent)

NOTES ON ALTERNATE ACCESS:

If the **alt_acc_pres** flag is set for a **MV_INFO_REQ_INFO** member and :

1. the primitive read request information specifies a list of variables, and
2. the corresponding primitive information **VARIABLE_LIST** structure has the **alt_access_pres** flag set to **SD_FALSE**,

then the Virtual Machine function will create the ASN.1 alternate access for the user and insert it into the variable list element. This will be freed automatically after the primitive request has been issued. No further action is required. Note that you are ALWAYS responsible for supplying a valid **mv_info_req_info->alt_acc** member, regardless of how the ASN.1 Alternate Access is created.

In creating the ASN.1 data elements, data will be encoded as before, except that if the **mv_info_req_info->alt_acc_pres** flag is **SD_TRUE**, the alternate access specified by the **alt_acc** element will be used in the conversion from local format to ASN.1. The **mv_info_req_info-> alt_acc_data_packed** flag will then determine whether the local data is assumed to be packed or not for this element.

After the return from this function, all information may be freed if desired; no commitment is required.

mv_info_nvlist

Usage: This function will send an InformationReport for the NamedVariableList object referenced by **`obj`**. It takes care of getting the data for each variable in the NamedVariableList, encoding the request, and sending it using **`mp_info`**.

Function Prototype: `ST_RET mv_info_nvlist (ST_INT chan, OBJECT_NAME *obj);`

Parameters:

`chan`	This indicates the channel on which to send InformationReport request.
`obj`	This pointer to structure of type **`OBJECT_NAME`** contains the NamedVariableList object.

Return Value: `ST_RET`

`SD_SUCCESS`	No Error.
`<> 0`	Error. (InformationReport not sent)

ms_extract_info_data

Usage: This support function extracts the data corresponding to the **i**th variable (where **i** is the index) from an InformationReport PDU. It converts it from the ASN.1 data representation (using a named type already existing in the MMS-EASE database), and puts the data into the user-specified memory address. Any AlternateAccess present for the **i**th variable is converted from its ASN.1 form and applied to the derived type. Use this function, within the **u_info_ind** function, to extract the variable information contained in an InformationReport. See the **ms_extract_varname** function for information on determining the names of the variables included in an InformationReport PDU, when named variables are used.

Function Prototype:

```
ST_RET ms_extract_info_data (MMSREQ_IND *ind,
                             ST_INT index,
                             NAMED_TYPE *type,
                             ST_CHAR *dest,
                             ST_BOOLEAN alt_acc_packed);
```

Parameters:

`ind` This pointer to the request control data structure of type **MMSREQ_IND** is passed to the **u_info_ind** function when it was called.

`index` This is an index indicating which variable, out of the list of variables contained in the InformationReport PDU, for which the data is to be extracted and converted. If this parameter equals 9, then the 10th variable is extracted and converted. The first variable in the list of variables would have an index of zero (0).

`type` This pointer to the **NAMED_TYPE** structure contains the named type definition existing in the MMS-EASE database. Use **ms_find_named_type_obj** to obtain this pointer.

`dest` This is a pointer to where you want to put the converted data. It should point to a C variable or structure corresponding to the type definition for the variable being extracted.

`alt_acc_packed` This flag indicates whether the data is to be packed when alternate access is used.

Return Value: `ST_RET` **SD_SUCCESS**. No error.

`<> 0` Error Code.

NOTES ON ALTERNATE ACCESS:

This function now handles alternate access when the indication is a list of variables. In this case, the Virtual Machine function uses the received ASN.1 alternate access in the conversion to local format.

The **alt_acc_data_packed** parameter then determines whether the local data is assumed to be packed or not for this element. In prior versions of this function, this parameter was ignored.

NOTE: This does not provide a good solution for a VariableList write where the VariableList is made up of variables with associated alternate access. For this situation, use the lower level data conversion functions to get the desired results.

ms_extract_inf_data

Usage: This support function extracts the data corresponding to the **i**th variable (where **i** is the index) from an InformationReport PDU. It converts it from the ASN.1 data representation (using a named type already existing in the MMS-EASE database), and puts the data into the user-specified memory address. Any AlternateAccess present for the **i**th variable is converted from it's ASN.1 form and applied to the derived type. The application will be returned the decoded AlternateAccess when the **alt_acc_ptr** is not passed in as a null.Use this function, within the **u_info_ind** function, to extract the variable information contained in an InformationReport. See the **ms_extract_varname** function for information on determining the names of the variables included in an InformationReport PDU, when named variables are used.

Function Prototype:

```
ST_RET ms_extract_inf_data (MMSREQ_IND *indptr,
                            ST_INT i,
                            NAMED_TYPE *type,
                            ST_CHAR *dest,
                            ST_BOOLEAN alt_acc_packed,
                            ALT_ACCESS *alt_acc_ptr);
```

Parameters:

`indptr` This pointer to the request control data structure of type **MMSREQ_IND** is passed to the **u_info_ind** function when it was called.

`i` This is an index indicating which variable, out of the list of variables contained in the InformationReport PDU, for which the data is to be extracted and converted. If this parameter equals 9, then the 10th variable is extracted and converted. The first variable in the list of variables would have an index of zero (0).

`type` This pointer to the **NAMED_TYPE** structure contains the named type definition existing in the MMS-EASE database. Use **ms_find_named_type_obj** to obtain this pointer.

`dest` This is a pointer to where you want to put the converted data. It should point to a C variable or structure corresponding to the type definition for the variable being extracted.

`alt_acc_packed` This flag indicates whether the data is to be packed when alternate access is used.

`alt_acc_ptr` This is an input and output parameter. The function will always decode and apply AlternateAccess to the derived type.

As an input parameter, it tells the function if it should return the decoded AlternateAccess to the application. If the **alt_acc_ptr** is supplied as a null to the function, no AlternateAccess information will be returned.

As an output parameter, this is the address of an **ALT_ACCESS** structure that contains the decoded alternate access definition when alternate access is present in the InfoReport service request. When AlternateAccess information is returned, it is the reponsibilty of the application to free the dynamic memory involved, for examply by using **chk_free**.

Return Value: `ST_RET` **SD_SUCCESS**. No error.

`<> 0` Error Code.

7. GetVariableAccessAttributes Service

This service is used to request that a VMD return the attributes of a Named Variable or an Unnamed Variable object defined at the VMD. Also, it can be used to request that a VMD return the derived type description of a Scattered Access object defined at the VMD.

Primitive Level GetVariableAccessAttributes Operations

The following section contains information on how to use the paired primitive interface for the GetVariableAccessAttributes service. This section covers available data structures used by the PPI, and the four primitive level functions that together make up the GetVariableAccessAttributes service. See **Volume 1 — Module 2 — General Sequence of Events — Confirmed Services** for a detailed explanation of the sequence of events that occur within the MMS-EASE environment during confirmed services.

- The GetVariableAccessAttributes service consists of the paired primitive functions of **`mp_getvar`**, **`u_getvar_ind`**, **`mp_getvar_resp`**, and **`u_mp_getvar_conf`**.

Data Structures

Request/Indication

The operation-specific data structure described below is used by the Client in issuing a GetVariableAccessAttributes request (**`mp_getvar`**). It is received by the Server when a GetVariableAccessAttributes indication (**`u_getvar_ind`**) is received.

```
struct getvar_req_info
  {
  ST_INT16     req_tag;
  OBJECT_NAME  name;
  VAR_ACC_ADDR address;
  };
typedef struct getvar_req_info GETVAR_REQ_INFO;
```

Fields:

`req_tag` This specifies the kind of variable:

`GETVAR_NAME`. This indicates a Named Variable.
`GETVAR_ADDR`. This indicates an Unnamed Addressed Variable.

`name` This structure of type **`OBJECT_NAME`** contains the name of the variable and is used only if **`req_tag = GETVAR_NAME`**.

`address` This structure of type **`VAR_ACC_ADDR`** indicates the address of the unnamed variable object and is used only if **`req_tag = GETVAR_ADDR`**.

NOTE: See the description in **Volume 1 — Module 2 — MMS Object Name Structure** for more information on **`OBJECT_NAME`** structure and page 2-150 for information on the **`VAR_ACC_ADDR`** structure.

Response/Confirm

The operation-specific data structure described on the next page is used by the Server in issuing a GetVariableAccessAttributes response (**mp_getvar_resp**). It is received by the Client when a GetVariableAccessAttributes confirm (**u_mp_getvar_conf**) is received.

```
struct getvar_resp_info
  {
  ST_BOOLEAN    mms_deletable;
  ST_BOOLEAN    address_pres;
  VAR_ACC_ADDR  address;
  VAR_ACC_TSPEC type_spec;
  };
typedef struct getvar_resp_info  GETVAR_RESP_INFO;
```

Fields:

`mms_deletable` — **SD_FALSE**. The variable definition is NOT deletable using a MMS service request.

SD_TRUE. The variable definition is deletable using a MMS service request.

`address_pres` — **SD_FALSE**. Do Not include **address** in the PDU.

SD_TRUE. Include **address** in the PDU. You should only include the address field if the variable is a NAMED variable, and access to it is PUBLIC.

`address` — This structure of type **VAR_ACC_ADDR** contains the address information for the specified public named variable.

`type_spec` — This structure of type **VAR_ACC_TSPEC** contains the type definition for the specified variable.

NOTE: See the description of the variable access data structures starting on page 2-150 for more information on these structures.

Paired Primitive Interface Functions

mp_getvar

Usage: This primitive request function sends a GetVariableAccessAttributes request PDU using the data from a structure of type **GETVAR_REQ_INFO**, pointed to by **info**. This service obtains the variable access attributes for a variable existing at the remote application.

Function Prototype:

```
MMSREQ_PEND *mp_getvar (ST_INT chan,
                        GETVAR_REQ_INFO *info);
```

Parameters:

chan	This is the channel number over which the request PDU is to be sent.
info	This pointer to structure of type **GETVAR_REQ_INFO** contains information specific to this request.

Return Value:

MMSREQ_PEND *	This pointer to the request control data structure of type **MMSREQ_PEND** is used to send the request PDU. In case of an error, the pointer is set to null and **mms_op_err** is written with the error code.

Corresponding User Confirmation Function: **u_mp_getvar_conf**

Operation-Specific Data Structure Used: **GETVAR_REQ_INFO**

See page 2-205 for a detailed description of this structure.

u_getvar_ind

Usage: This user function is called when a GetVariableAccessAttributes indication is received by a Server node. The contents and operation of this function are user-defined, and depend on the application requirements. To conform to the MMS protocol requirement, the application must do one of the following some time after the indication is received:

1) send a positive response using either:

   a) the primitive response function (**`mp_getvar_resp`**) after obtaining the specified variable's attributes,

   b) the virtual machine response function (**`mv_getvar_resp`**) to have the virtual machine take the requested action and send the response, or

2) call the appropriate primitive negative (error) response function (**`mp_err_resp`**). See **Volume 3 — Module 11 — MMS-EASE Error Handling** for an explanation of this function.

See **Volume 1 — Module 1 — User Indication Function Class** for additional information regarding the handling of indications.

Function Prototype: `ST_VOID u_getvar_ind (MMSREQ_IND *ind);`

Parameters:

`ind` This pointer to the **`MMSREQ_IND`** structure is described in **Volume 1 — Module 2 — Request and Indication Control Data Structures**. This pointer must be used to send the response to this indication. The primary item of interest in this structure is **`ind->req_info_ptr`**. This is a pointer to the operation-specific data structure for a GetVariableAccessAttributes indication (**`GETVAR_REQ_INFO`**). This pointer will always be valid when **`u_getvar_ind`** is called.

Return Value: `ST_VOID` (ignored)

Operation-Specific Data Structure Used: **`GETVAR_REQ_INFO`**

See page 2-205 for a detailed description of this structure.

mp_getvar_resp

Usage: This primitive response function sends a GetVariableAccessAttributes positive response PDU. This should be called after the **u_getvar_ind** function is called (a GetVariableAccessAttributes indication is received), and after the specified variable access attribute information has been successfully obtained.

Function Prototype:

```
ST_RET mp_getvar_resp (MMSREQ_IND *ind,
                       GETVAR_RESP_INFO *info);
```

Parameters:

`ind` This pointer to the indication control structure of type **MMSREQ_IND** is received in the indication service function, **u_getvar_ind**.

`info` This pointer to an Operation-Specific data structure of type **GETVAR_RESP_INFO** contains information specific to the GetVariableAccessAttributes response PDU to be sent.

Return Value: `ST_RET` **SD_SUCCESS**. No Error.

`<> 0` Error Code.

Corresponding User Indication Function: **u_getvar_ind**

Operation-Specific Data Structure Used: **GETVAR_RESP_INFO**

See page 2-206 for a detailed description of this structure.

u_mp_getvar_conf

Usage: This primitive user confirm function is called when the confirm to a GetVariableAccess request (**mp_getvar**) is received. **resp_err** variable contains a value indicating whether an error occurred. **resp_info_ptr** may contain data that the remote node sent back in the confirmation or it may contain error information if an error occurred. If no error has occurred (**req->resp_err = CNF_RESP_OK**), Get Variable Access Attribute information is available to the application using the **req->resp_info_ptr** pointer. See **Volume 1 — Module 1 — Primitive User Confirmation Class** for specific information on the recommended handling of confirms.

Note that the function called for this event may be changed if desired; however, **u_mp_getvar_conf** is the standard default function.

Function Prototype:

```
ST_VOID u_mp_getvar_conf (MMSREQ_PEND *req);
```

Parameters:

`req` This pointer to the **MMSREQ_PEND** structure is returned from the original **mp_getvar** request function. See **Volume 1 — Module 2 — Request and Indication Control Data Structures** for more information on this structure.

For a positive response (**req->resp_err = CNF_RESP_OK**), the request control structure member **resp_info_ptr** is a pointer to the operation-specific data structure (**GETVAR_RESP_INFO**) for the GetVariableAccessAttributes function.

For a negative response, or other errors, see **Volume 3 — Module 11 — MMS-EASE Error Handling** for more information.

Return Value: `ST_VOID` (ignored)

Operation-Specific Data Structure Used: **GETVAR_RESP_INFO**

See page 2-206 for a detailed description of this structure.

Virtual Machine GetVariableAccessAttributes Operations

The following section contains information on how to use the virtual machine interface for the GetVariableAccessAttributes service. It covers the function that makes up the VMI GetVariableAccessAttributes service.

- The GetVariableAccessAttributes service consists only of the virtual machine function of **`mv_getvar_resp`**.

There are no virtual machine data structures regarding this service.

Virtual Machine Interface Function

mv_getvar_resp

Usage: This virtual machine response function allows a user to search through the MMS-EASE database and automatically send a GetVariableAccessAttributes positive response PDU. Currently only named variables are supported. This function will return an error if other than a named variable is used, or if the VMD object is undefined. BE SURE TO SET **`mms_chan_info.objs.vmd`** TO POINT TO THE PROPER VMD STRUCTURE BEFORE CALLING THIS FUNCTION. See **Volume 1 — Module 2 — Channel Information Structure** and **Module 4 — VMD Control** starting on page 2-10 for more information.

Function Prototype: `ST_RET mv_getvar_resp (MMSREQ_IND *ind);`

Parameters:

`ind` This pointer to the request control data structure of type **`MMSREQ_IND`** is received in the indication service function, **`u_getvar_ind`**.

Return Value: `ST_RET` **`SD_SUCCESS`**. No Error.

`<> 0` Error Code.

Corresponding User Indication Function: **`u_getvar_ind`**

See page 2-208 for information on this function.

8. DefineNamedVariable Service

This service is used by a Client application to request that a Server VMD create a Named Variable object, which describes mapping to a real variable in the VMD.

Primitive Level DefineNamedVariable Operations

The following section contains information on how to use the paired primitive interface for the DefineNamed-Variable service. This section covers available data structures used by the PPI, and the four primitive level functions that together make up the DefineNamedVariable service. See **Volume 1 — Module 2 — General Sequence of Events — Confirmed Services** for a detailed explanation of the sequence of events that occur within the MMS-EASE environment during confirmed services.

- The DefineNamedVariable service consists of the paired primitive functions of **mp_defvar**, **u_defvar_ind**, **mp_defvar_resp**, and **u_mp_defvar_conf**.

Data Structures

Request/Indication

The operation-specific data structure described below is used by the Client in issuing a DefineNamedVariable request (**mp_defvar**). It is received by the Server when a DefineNamedVariable indication (**u_defvar_ind**) is received.

```
struct defvar_req_info
  {
  OBJECT_NAME   name;
  VAR_ACC_ADDR  address;
  ST_BOOLEAN    type_spec_pres;
  VAR_ACC_TSPEC type_spec;
  };
typedef struct defvar_req_info DEFVAR_REQ_INFO;
```

Fields:

`name` This structure of type **OBJECT_NAME** contains the name of the Named Variable to be defined.

`address` This structure of type **VAR_ACC_ADDR** contains the address of the variable to be defined.

`type_spec_pres` **SD_FALSE**. Do Not include **type_spec** in PDU.

SD_TRUE. Include **type_spec** in PDU.

`type_spec` This structure of type **VAR_ACC_TSPEC** indicates the Type Specification for the Named Variable to be defined.

NOTE: See the description in **Volume 1 — Module 2 — MMS Object Name Structure** for information on **OBJECT_NAME** and page 2-150 for more information on the **VAR_ACC_ADDR**, and **VAR_ACC_TSPEC** structures.

Paired Primitive Interface Functions

mp_defvar

Usage: This primitive request function sends a DefineNamedVariable request PDU using the data from a structure of type **DEFVAR_REQ_INFO**, pointed to by **info**. This service is used to store a named variable specification for use in subsequent access to that named variable at the remote node.

Function Prototype:
```
MMSREQ_PEND *mp_defvar (ST_INT chan,
                        DEFVAR_REQ_INFO *info);
```

Parameters:

chan	This is the channel number over which the request PDU is to be sent.
info	This pointer to a structure of type **DEFVAR_REQ_INFO** contains information specific to this request.

Return Value:

MMSREQ_PEND *	This pointer to the request control data structure of type **MMSREQ_PEND** is used to send the request PDU. In case of an error, the pointer is set to null and **mms_op_err** is written with the error code.

Corresponding User Confirmation Function: **u_mp_defvar_conf**

Operation-Specific Data Structure Used: **DEFVAR_REQ_INFO**

See page 2-213 for a detailed description of this structure.

u_defvar_ind

Usage: This user function is called when a DefineNamedVariable indication is received by a Server node. The contents and operation of this function are user-defined and depend on the application requirements. To conform to the MMS protocol requirements, the application must do one of the following some time after the indication is received:

1) send a positive response using either:

   a) the primitive response function (**`mp_defvar_resp`**) after the Named Variable object has been created,

   b) the virtual machine response function (**`mv_defvar_resp`**) to have the virtual machine take the requested action and send the response, or

2) call the appropriate primitive negative (error) response function (**`mp_err_resp`**). See **Volume 3 — Module 11 — MMS-EASE Error Handling** for an explanation of this function.

See **Volume 1 — Module 1 — User Indication Function Class** for additional information regarding the handling of indications.

Function Prototype: `ST_VOID u_defvar_ind (MMSREQ_IND *ind);`

Parameters:

`ind` This pointer to the **MMSREQ_IND** structure is described in **Volume 1 — Module 2 — Request and Indication Control Data Structures**. This pointer must be used to send the response to this indication. The primary item of interest in this structure is **`ind->req_info_ptr`**. This is a pointer to the operation-specific data structure (**`DEFVAR_REQ_INFO`**) for a Define Named Variable indication. This pointer will always be valid when **`u_defvar_ind`** is called.

Return Value: `ST_VOID` (ignored)

Operation-Specific Data Structure Used: **`DEFVAR_REQ_INFO`**

See page 2-213 for a detailed description of this structure.

mp_defvar_resp

Usage: This primitive response function sends a DefineNamedVariable positive response PDU. This function should be called as a response to the **u_defvar_ind** function being called (a DefineNamedVariable indication is received), and after the specified variable name was successfully created.

Function Prototype: `ST_RET mp_defvar_resp (MMSREQ_IND *ind);`

Inputs:

`ind` This pointer to the indication control structure of type **MMSREQ_IND** is received in the indication service function, **u_defvar_ind**.

Return Value: `ST_RET` **SD_SUCCESS**. No Error.

`<> 0` Error Code.

Corresponding User Indication Function: **u_defvar_ind**

Operation-Specific Data Structure Used: NONE

u_mp_defvar_conf

Usage: This primitive user confirm function is called when a confirm to a DefineNamedVariable request (**mp_defvar**) is received. **resp_err** variable contains a value indicating whether an error occurred. **resp_info_ptr** may contain data that the remote node sent back in the confirmation or it may contain error information if an error occurred. If no error has occurred, (**req->resp_err = CNF_RESP_OK**), DefineNamedVariable information is available to the application using the **req->resp_info_ptr** pointer. See **Volume 1 — Module 1 — Primitive User Confirmation Function Class** for specific information on the recommended handling of confirms.

Note that the function called for this event may be changed, if desired; however, **u_mp_defvar_conf** is the standard default function.

Function Prototype:

```
ST_VOID u_mp_defvar_conf (MMSREQ_PEND *req);
```

Parameters:

`req` This pointer of structure type **MMSREQ_PEND** is returned from the original **mp_defvar** function. See **Volume 1 — Module 2 — Request and Indication Control Data Structures** for more information on this structure.

For a positive response (**req->resp_err = CNF_RESP_OK**), the request control structure member **resp_info_pres** indicates that there is no confirm data for this function.

For a negative response, or other errors, see **Volume 3 — Module 11 — MMS-EASE Error Handling** for more information.

Return Value: `ST_VOID` (ignored)

Operation-Specific Data Structure Used: NONE

Virtual Machine DefineNamedVariable Operations

The following section contains information on how to use the virtual machine interface for the DefineNamed-Variable service.

- The DefineNamedVariable service consists of the virtual machine function of **`mv_defvar_resp`**.

There are no virtual machine data structures regarding this service.

Virtual Machine Interface Function

mv_defvar_resp

Usage: This virtual machine response function allows a user to respond to a Define Named Variable indication (**`u_defvar_ind`**). This function will attempt to add the variable to the local Virtual Machine Database of named variables. BE SURE TO SET **`mms_chan_info.objs.vmd`** TO POINT TO THE PROPER VMD STRUCTURE BEFORE CALLING THIS FUNCTION. See **Volume 1 — Module 2 — Channel Information Structure** and **Module 4 — VMD Control** starting on page 2-10 for more information.

Function Prototype: `ST_RET mv_defvar_resp (MMSREQ_IND *ind);`

Parameters:

`ind` This pointer to the structure of type **`MMSREQ_IND`** is received for indication service function, **`u_defvar_ind`**.

Return Value: `ST_RET` **`SD_SUCCESS`**. No Error.

`<> 0` Error Code.

Corresponding User Indication Function: **`u_defvar_ind`**

9. DeleteVariableAccess Service

This service is used by a Client application to request that a VMD delete one or more Named Variables or Scattered Access Objects having a MMS Deletable attribute equal to true.

Primitive Level DeleteVariableAccess Operations

The following section contains information on how to use the paired primitive interface for the DeleteVariableAccess service. This section covers available data structures used by the PPI, and the four primitive level functions that together make up the DeleteVariableAccess service. See **Volume 1 — Module 2 — General Sequence of Events — Confirmed Services** for a detailed explanation of the sequence of events that occur within the MMS-EASE environment during confirmed services.

- The DeleteVariableAccess service consists of the paired primitive functions of **`mp_delvar`**, **`u_delvar_ind`**, **`mp_delvar_resp`**, and **`u_mp_delvar_conf`**.

Data Structures

Request/Indication

The operation-specific data structure described below is used by the Client in issuing a DeleteVariableAccess request (**`mp_delvar`**). It is received by the Server when a DeleteVariableAccess indication (**`u_delvar_ind`**) is received.

```
struct delvar_req_info
  {
  ST_INT16    scope;
  ST_BOOLEAN  dname_pres;
  ST_CHAR     dname [MAX_IDENT_LEN+1];
  ST_BOOLEAN  vnames_pres;
  ST_INT      num_of_vnames;
/*OBJECT_NAME vname_list [num_of_vnames];                          */
  SD_END_STRUCT
  };
typedef struct delvar_req_info DELVAR_REQ_INFO;
```

Fields:

`scope` This indicates the scope of the named variable definition(s) to be deleted:

`DELVAR_SPEC`. Delete only those variables whose names are in **`vname_list`**.

`DELVAR_AA`. The variables are specific to this association (aa-specific). Delete all aa-specific variables.

`DELVAR_DOM`. Delete all domain specific variables in the specified domain (**`dname`**).

`DELVAR_VMD`. Delete all VMD-Specific variables.

`dname_pres` **`SD_FALSE`**. Do Not include **`dname`** in the PDU.

`SD_TRUE`. Include **`dname`** in the PDU.

`dname` This contains the name of the domain for which all domain specific variables are to be deleted. Use if **`scope = DELVAR_DOM`**.

vnames_pres	**SD_FALSE**. Do Not include **vnames** in the PDU.
	SD_TRUE. Include **vnames** in the PDU.
num_of_vnames	This indicates the number of variables to be deleted.
vname_list	This array of structures of type **OBJECT_NAME** indicates the specific variables to be deleted. See the description in **Volume 1 — Module 2 — MMS Object Name Structure** for more information on this structure.

NOTE:

1. FOR REQUEST ONLY, when allocating a data structure of type **DELVAR_REQ_INFO**, enough memory must be allocated to hold the information for the **vname_list** member of the structure. The following C statement can be used:

```
info = (DELVAR_REQ_INFO *) chk_malloc (sizeof(DELVAR_REQ_INFO) + (num_of_vnames *
            (sizeof(OBJECT_NAME))));
```

Response/Confirm

The operation-specific data structure described below is used by the Server in issuing a DeleteVariableAccess response (**mp_delvar_resp**). It is received by the Client when a Delete Variable Access confirm (**u_mp_delvar_conf**) is received.

```
struct delvar_resp_info
  {
  ST_UINT32  num_matched;
  ST_UINT32  num_deleted;
  };
typedef struct delvar_resp_info  DELVAR_RESP_INFO;
```

Fields:

num_matched	This indicates the number of variable descriptions specified in the request that matched an existing variable.
num_deleted	This indicates the number of variables actually deleted.

Paired Primitive Interface Functions

mp_delvar

Usage: This primitive request function is used to send a DeleteVariableAccess request PDU using the data from a structure of type **DELVAR_REQ_INFO**, pointed to by **info**. This service is used to delete one or more Named Variables or Scattered Accesses at the remote node. These variables were previously defined via a DefineNamedVariable or DefineScatteredAccess service request.

Function Prototype:
```
MMSREQ_PEND *mp_delvar (ST_INT chan,
                        DELVAR_REQ_INFO *info);
```

Parameters:

`chan` This is the channel number over which the request PDU is to be sent.

`info` This pointer to structure of type **DELVAR_REQ_INFO** contains information specific to this request.

Return Value:

`MMSREQ_PEND *` This pointer to the request control data structure of type **MMSREQ_PEND** is used to send the request PDU. In case of an error, the pointer is set to null and **mms_op_err** is written with the error code.

Corresponding User Confirmation Function: **u_mp_delvar_conf**

Operation-Specific Data Structure Used: **DELVAR_REQ_INFO**

See page 2-219 for a detailed description of this structure.

u_delvar_ind

Usage: This user function is called when a DeleteVariableAccess indication is received by a Server node. The contents and operation of this function are user-defined, and depend on the application requirements. To conform to the MMS protocol requirements, the application must do one of the following some time after the indication is received:

1) send a positive response using either:

   a) the primitive response function (**mp_delvar_resp**) after the Named Variable or Scattered Access object has been deleted,

   b) the virtual machine response function (**mv_delvar_resp**) to have the virtual machine take the requested action and send the response, or

2) call the appropriate primitive negative (error) response function (**mp_err_resp**). See **Volume 3 — Module 11 — MMS-EASE Error Handling** for an explanation of this function.

See **Volume 1 — Module 1 — User Indication Function Class** for additional information regarding the handling of indications.

Function Prototype: `ST_VOID u_delvar_ind (MMSREQ_IND *ind);`

Parameters:

`ind` This pointer to the **MMSREQ_IND** structure is described in **Volume 1 — Module 2 — Request and Indication Control Data Structures**. This pointer must be used to send the response to this indication. The primary item of interest in this structure is **ind->req_info_ptr**. This is a pointer to the operation-specific data structure (**DELVAR_REQ_INFO**). This pointer will always be valid when **u_delvar_ind** is called.

Return Value: `ST_VOID` (ignored)

Operation-Specific Data Structure Used: **DELVAR_REQ_INFO**

See page 2-219 for a detailed description of this structure.

mp_delvar_resp

Usage: This primitive response function sends a DeleteVariableAccess positive response PDU. This should be called as a response to the **u_delvar_ind** function being called (a DeleteVariableAccess indication is received), and after the specified named variable or scattered access was successfully deleted. Variables should only be deleted that have the MMS deletable attribute set. If a request to delete more than one variable is made and at least one of the variables is not MMS deletable while the others were deleted, a positive response should still be sent indicating how many variables were deleted. If any of the specified variables cannot be deleted for any reason other than a conflict with the MMS deletable attribute, an error response should be sent using **mp_error_resp** and the additional error information specifying how many of the variables were actually deleted.

Function Prototype:

```
ST_RET mp_delvar_resp (MMSREQ_IND *ind,
                       DELVAR_RESP_INFO *info);
```

Parameters:

`ind` This pointer to the indication control structure of type **MMSREQ_IND** is received in the indication service function, **u_delvar_ind**.

`info` This pointer to an Operation-Specific data structure of type **DELVAR_RESP_INFO** contains information specific to the Delete Variable Access response PDU to be sent.

Return Value: `ST_RET` **SD_SUCCESS**. No Error.

`<> 0` Error Code.

Corresponding User Indication Function: **u_delvar_ind**

Operation-Specific Data Structure Used: **DELVAR_RESP_INFO**

See page 2-220 for a detailed description of this structure.

u_mp_delvar_conf

Usage: This primitive user confirm function is called when the confirm to a DeleteVariableAccess request (**mp_delvar**) is received. **resp_err** contains a value indicating whether an error occurred. **resp_info_ptr** may contain data that the remote node sent back in the confirmation, or it may contain error information if an error occurred. If no error has occurred, (**req->resp_err = CNF_RESP_OK**), Delete Variable Access information is available to the application using the **req->resp_info_ptr** pointer. See **Volume 1 — Module 1 — Primitive User Confirmation Function Class** for specific information regarding recommended handling of confirms.

Note that the function called for this event may be changed if desired; however, **u_mp_delvar_conf** is the standard default function.

Function Prototype:

```
ST_VOID u_mp_delvar_conf (MMSREQ_PEND *req);
```

Parameters:

`req` This pointer to the **MMSREQ_PEND** structure is returned from the original **mp_delvar** request function. See **Volume 1 — Module 2 — Request and Indication Control Data Structures** for more information on this structure.

For positive response (**req->resp_err = CNF_RESP_OK**), the request control structure member **resp_info_ptr** is a pointer to the operation-specific data structure (**DELVAR_RESP_INFO**) for the DeleteVariableAccess function.

For a negative response, or other errors, see **Volume 3 — Module 11 — MMS-EASE Error Handling** for more information.

Return Value: `ST_VOID` (ignored)

Operation-Specific Data Structure Used: **DELVAR_RESP_INFO**

See page 2-220 for a detailed description of this structure.

Virtual Machine DeleteVariableAccess Operations

The following section contains information on how to use the virtual machine interface for the DeleteVariableAccess service.

- The DeleteVariableAccess service consists of the virtual machine function of **mv_delvar_resp**.

There are no virtual machine data structures concerning this service.

Virtual Machine Interface Function

mv_delvar_resp

Usage: This virtual machine response function attempts to delete the specified variable(s) from the local Virtual Machine Database. It also responds to a DeleteVariableAccess indication (**u_delvar_ind**). This function automatically returns an error if there is a problem in deleting a variable (for other than a not-Deletable attribute). BE SURE TO SET **mms_chan_info.objs.vmd** TO POINT TO THE PROPER VMD STRUCTURE BEFORE CALLING THIS FUNCTION. See **Volume 1 — Module 2 — Channel Information Structure** and **Module 4 — VMD Control** starting on page 2-10 for more information.

Function Prototype: `ST_RET mv_delvar_resp (MMSREQ_IND *ind);`

Parameters:

`ind` This pointer to the structure of type **MMSREQ_IND** is received for indication service function, **u_delvar_ind**.

Return Value: `ST_RET` **SD_SUCCESS**. No Error.

`<> 0` Error Code.

Corresponding User Indication Function: **u_delvar_ind**

10. DefineNamedVariableList Service

This service is used by a Client application to request that a Server VMD create a Named Variable List object. This allows access through a list of Named Variable objects, Unnamed Variable objects, or Scattered Access objects, or any combination.

Primitive Level DefineNamedVariableList Operations

The following section contains information on how to use the paired primitive interface for the DefineNamedVariableList service. This section covers available data structures used by the PPI, and the four primitive level functions that together make up the DefineNamedVariableList service. See **Volume 1 — Module 2 — General Sequence of Events — Confirmed Services** for a detailed explanation of the sequence of events that occur within the MMS-EASE environment during confirmed services.

- The DefineNamedVariableList service consists of the paired primitive functions of **`mp_defvlist`**, **`u_defvlist_ind`**, **`mp_defvlist_resp`**, and **`u_mp_defvlist_conf`**.

Data Structures

Request/Indication

The operation-specific data structure described below is used by the Client in issuing a DefineNamedVariableList request (**`mp_defvlist`**). It is received by the Server when a DefineNamedVariableList indication (**`u_defvlist_ind`**) is received.

```
struct defvlist_req_info
  {
  OBJECT_NAME   vl_name;
  ST_INT        num_of_variables;
/*VARIABLE_LIST var_list [num_of_variables];              */
  SD_END_STRUCT
  };
typedef struct defvlist_req_info DEFVLIST_REQ_INFO;
```

Fields:

`vl_name` This structure of type **`OBJECT_NAME`** contains the name of the variable list to be defined.

`num_of_variables` This indicates the number of variables in this list.

`var_list` This array of structures of type **`VARIABLE_LIST`** contains the variable descriptions for the list of variables to be accessed.

NOTES:

1. See the description in **Volume 1 — Module 2 — MMS Object Name Structure** for more information on the **`OBJECT_NAME`** structure and page 2-150 for information on the **`VARIABLE_LIST`** structure.
2. FOR REQUEST ONLY, when allocating Operation-Specific data structures containing a structure of type **`VARIABLE_LIST`**, make sure that sufficient memory is allocated to hold the list of variables contained in **`var_list`**. The following C Statement can be used:

```
info = (DEFVLIST_REQ_INFO *) chk_malloc (sizeof(DEFVLIST_REQ_INFO) +
            (num_of_variables * (sizeof(VARIABLE_LIST))));
```

Paired Primitive Interface Functions

mp_defvlist

Usage: This primitive request function sends a Define Named Variable List request PDU using the data from a structure of type **DEFVLIST_REQ_INFO**, pointed to by **info**. This service is used to define a named variable list object at a remote application.

Function Prototype:
```
MMSREQ_PEND *mp_defvlist (ST_INT chan,
                          DEFVLIST_REQ_INFO *info);
```

Parameters:

chan	This is the channel number over which the request PDU is to be sent.
info	This pointer to a structure of type **DEFVLIST_INFO** contains information specific to this request.

Return Value:

MMSREQ_PEND *	This pointer to the request control data structure of type **MMSREQ_PEND** is used to send the request PDU. In case of an error, the pointer is set to null and **mms_op_err** is written with the error code.

Corresponding User Confirmation Function: **u_mp_defvlist_conf**

Operation-Specific Data Structure Used: **DEFVLIST_REQ_INFO**

See page 2-227 for a detailed description of this structure.

u_defvlist_ind

Usage: This user function is called when a DefineNamedVariableList indication is received by a Server node. The contents and operation of this function are user-defined, and depend on the application. To conform to the MMS protocol requirements, the application must do one of the following some time after the indication is received:

1) send a positive response using the primitive response function (**`mp_defvlist_resp`**) after the Named Variable List has been created, or

2) call the appropriate primitive negative (error) response function (**`mp_err_resp`**). See **Volume 3 — Module 11 — MMS-EASE Error Handling** for an explanation of this function.

See **Volume 1 — Module 1 — User Indication Function Class** for additional information regarding the handling of indications.

Function Prototype: `ST_VOID u_defvlist_ind (MMSREQ_IND *ind);`

Parameters:

`ind` This pointer to the **`MMSREQ_IND`** structure is described in **Volume 1 — Module 2 — Request and Indication Control Data Structures**. This pointer must be used to send the response to this indication. The primary item of interest in this structure is **`ind->req_info_ptr`**. This is a pointer to the operation-specific data structure (**`DEFVLIST_REQ_INFO`**) for a Define Named Variable List indication. This pointer will always be valid when **`u_defvlist_ind`** is called.

Return Value: `ST_VOID` (ignored)

Operation-Specific Data Structure Used: **`DEFVLIST_REQ_INFO`**

See page 2-227 for a detailed description of this structure.

mp_defvlist_resp

Usage: This primitive response function sends a DefineNamedVariableList positive response PDU. This function should be called as a response to the **u_defvlist_ind** function being called (a DefineNamedVariableList indication), and after the specified variable list was successfully created.

Function Prototype: `ST_RET mp_defvlist_resp (MMSREQ_IND *info);`

Parameters:

`info` This pointer to the indication control structure of type **MMSREQ_IND** is received in the indication service function, **u_defvlist_ind**.

Return Value: `ST_RET` **SD_SUCCESS**. No Error.

`<> 0` Error Code.

Corresponding User Indication Function: **u_defvlist_ind**

Operation-Specific Data Structure Used: NONE

u_mp_defvlist_conf

Usage: This primitive user confirm function is called when the confirm to a DefineNamedVariable List request (**mp_defvlist**) is received. **resp_err** contains a value indicating whether an error occurred. **resp_info_ptr** may contain data that the remote node sent back in the confirmation or it may contain error information if an error occurred. If no error has occurred (**req->resp_err = CNF_RESP_OK**), DefineNamedVariableList information is available to the application using the **req->resp_info_ptr** pointer. See **Volume 1 — Module 1 — Primitive User Confirmation Function Class** for specific information on the recommended handling of confirms.

Note that the function called for this event may be changed if desired; however, **u_mp_defvlist_conf** is the standard default function.

Function Prototype:

```
ST_VOID u_mp_defvlist_conf (MMSREQ_PEND *req);
```

Parameters:

`req` This pointer to the **MMSREQ_PEND** structure is returned from the original **mp_defvlist** function. See **Volume 1 — Module 2 — Request and Indication Control Data Structures**.

For a positive response (**req->resp_err = CNF_RESP_OK**), the request control structure member **resp_info_pres** indicates that there is no confirm data for this function.

For a negative response, or other errors, see **Volume 3 — Module 11 — MMS-EASE Error Handling** for more information.

Return Value: `ST_VOID` (ignored)

Operation-Specific Data Structure Used: NONE

Virtual Machine DefineNamedVariableList Operations

The following section contains information on how to use the virtual machine interface for the DefineNamed-VariableList service.

- The DefineNamedVariableList service consists of the virtual machine function of **`mv_defvlist_resp`**.

There are no virtual machine data structures concerning this service.

Virtual Machine Interface Function

mv_defvlist_resp

Usage: This virtual machine response function allows a user to respond to a DefineNamedVariable List indication (**`u_defvlist_ind`**). This function will attempt to add the variable list to the local Virtual Machine Database of named variable lists. BE SURE TO SET **`mms_chan_info.objs.vmd`** TO POINT TO THE PROPER VMD STRUCTURE BEFORE CALLING THIS FUNCTION. See **Volume 1 — Module 2 — Channel Information Structure** and **Module 4 — VMD Control** starting on page 2-10 for more information.

Function Prototype: `ST_RET mv_defvlist_resp (MMSREQ_IND *ind);`

Parameters:

`ind` This pointer to the structure of type **`MMSREQ_IND`** is received for indication service function, **`u_defvlist_ind`**.

Return Value: `ST_RET` **`SD_SUCCESS`**. No Error.

`<> 0` Error Code.

Corresponding User Indication Function: **`u_defvlist_ind`**

11. GetNamedVariableListAttributes Service

This service is used by a Client application to request that a Server VMD return the attributes of a Named Variable List object defined at the VMD.

Primitive Level GetNamedVariableListAttributes Operations

The following section contains information on how to use the paired primitive interface for the GetNamed-VariableListAttributes service. This section covers available data structures used by the PPI, and the four primitive level functions that together make up the GetNamedVariableListAttributes service. See **Volume 1 — Module 2 — General Sequence of Events — Confirmed Services** for a detailed explanation of the sequence of events that occur within the MMS-EASE environment during confirmed services.

- The GetNamedVariableListAttributes service consists of the paired primitive functions of **`mp_getvlist`**, **`u_getvlist_ind`**, **`mp_getvlist_resp`**, and **`u_mp_getvlist_conf`**.

Data Structures

Request/Indication

The operation-specific data structure described below is used by the Client in issuing a GetNamedVariableList Attributes request (**`mp_getvlist`**). It is received by the Server when a GetNamedVariableListAttributes indication (**`u_getvlist_ind`**) is received.

```
struct getvlist_req_info
  {
  OBJECT_NAME vl_name;
  };
typedef struct getvlist_req_info GETVLIST_REQ_INFO;
```

Fields:

`vl_name` This structure of type **`OBJECT_NAME`** contains the name of the variable list to be defined. See the description in **Volume 1 — Module 2 — MMS Object Name Structure** for more information on this structure.

Response/Confirm

The operation-specific data structure described below is used by the Server in issuing a GetNamedVariableList Attributes response (**`mp_getvlist_resp`**). It is received by the Client when a GetNamedVariableListAttributes confirm (**`u_mp_getvlist_conf`**) is received.

```
struct getvlist_resp_info
  {
  ST_BOOLEAN    mms_deletable;
  ST_INT        num_of_variables;
/*VARIABLE_LIST var_list [num_of_variables];              */
  SD_END_STRUCT
  };
typedef struct getvlist_resp_info GETVLIST_RESP_INFO;
```

Fields:

`mms_deletable` **SD_FALSE**. The variable list definition is NOT deletable using a MMS service request.

SD_TRUE. The variable list definition is deletable using a MMS service request.

`num_of_variables` This indicates the number of variables in this named variable list.

`var_list` This array of structures of type **VARIABLE_LIST** contains the variable descriptions for variables in the Named Variable List object. See note below on allocation exceptions.

NOTE: FOR RESPONSE ONLY, when allocating a data structure of type **GETVLIST_RESP_INFO**, enough memory must be allocated to hold the information for the **var_list** member of the structure. The following C statement can be used:

```
info = (GETVLIST_RESP_INFO*) chk_malloc (sizeof (GETVLIST_RESP_INFO) +
            (num_of_variables * (sizeof (VARIABLE_LIST))));
```

Paired Primitive Interface Functions

mp_getvlist

Usage: This primitive request function sends a GetNamedVariableListAttributes request PDU using the data from a structure of type **GETVLIST_REQ_INFO**, pointed to by **info**. This service obtains the attributes of a Named Variable List object existing at the remote application.

Function Prototype:

```
MMSREQ_PEND *mp_getvlist (ST_INT chan,
                          GETVLIST_REQ_INFO *info);
```

Parameters:

chan	This is the channel number over which the request PDU is to be sent.
info	This pointer to structure of type **GETVLIST_REQ_INFO** contains information specific to this request.

Return Value:

MMSREQ_PEND *	This pointer to the request data structure of type **MMSREQ_PEND** is used to send the request PDU. In case of an error, the pointer is set to null and **mms_op_err** is written with the error code.

Corresponding User Confirmation Function: **u_mp_getvlist_conf**

Operation-Specific Data Structure Used: **GETVLIST_REQ_INFO**

See page 2-233 for a detailed description of this structure.

u_getvlist_ind

Usage: This user function is called when a GetNamedVariableListAttributes indication is received by a Server node. The contents and operation of this function are user-defined, and depend on the application requirements. To conform to the MMS protocol requirements, the application must do one of the following some time after the indication is received:

1) send a positive response using the primitive response function (**`mp_getvlist_resp`**) after the attributes of the Named Variable List have been obtained, or

2) call the appropriate primitive negative (error) response function (**`mp_err_resp`**). Please refer to **Volume 3 — Module 11 — MMS-EASE Error Handling** for an explanation of this function.

See **Volume 1— Module 1 — User Indication Function Class** for additional information regarding the handling of indications.

Function Prototype:

```
ST_VOID u_getvlist_ind (MMSREQ_IND *ind);
```

Parameters:

`ind` This pointer to the **`MMSREQ_IND`** structure is described in **Volume 1 — Module 2 — Request and Indication Control Data Structures**. This pointer must be used to send the response to this indication. The primary item of interest in this structure is **`ind->req_info_ptr`**. This is a pointer to the operation-specific data structure (**`GETVLIST_REQ_INFO`**) for a GetNamedVariableListAttributes indication. This pointer will always be valid when **`u_getvlist_ind`** is called.

Return Value: `ST_VOID` (ignored)

Operation-Specific Data Structure Used: **`GETVLIST_REQ_INFO`**

See page 2-233 for a detailed description of this structure.

mp_getvlist_resp

Usage: This primitive response function sends a GetNamedVariableListAttributes positive response PDU. This should be called after the **u_getvlist_ind** function is called (a GetNamedVariableListAttributes indication is received), and after the specified named variable list attribute information has been successfully obtained.

Function Prototype:

```
ST_RET mp_getvlist_resp (MMSREQ_IND *ind,
                         GETVLIST_RESP_INFO *info);
```

Parameters:

`ind` This pointer to the indication control structure of type **MMSREQ_IND** is received in the **u_getvlist_ind** indication service function.

`info` This pointer to an Operation-Specific data structure of type **GETVLIST_RESP_INFO** contains information specific to the PDU to be sent.

Return Value: `ST_RET` **SD_SUCCESS**. No Error.

`<> 0` Error Code.

Corresponding User Indication Function: **u_getvlist_ind**

Operation-Specific Data Structure Used: **GETVLIST_RESP_INFO**

See page 2-233 for a detailed description of this structure.

u_mp_getvlist_conf

Usage: This primitive user confirm function is called when the confirm to a GetNamedVariableList request (**mp_getvlist**) is received. **resp_err** contains a value indicating whether an error occurred. **resp_info_ptr** may contain data that the remote node sent back in the confirmation or it may contain error information if an error occurred. If no error has occurred (**req->resp_err = CNF_RESP_OK**), GetNamedVariableListAttributes information is available to the application using the **req->resp_info_ptr** pointer. See **Volume 1 — Module 1 — Primitive User Confirmation Function Class** for specific information on the recommended handling of confirms.

Note that the function called for this event may be changed if desired; however, **u_mp_getvlist_conf** is the standard default function.

Function Prototype:

```
ST_VOID u_mp_getvlist_conf (MMSREQ_PEND *req);
```

Parameters:

`req` This pointer to the **MMSREQ_PEND** structure is returned from the original **mp_getvlist** request function. See **Volume 1 — Module 2 — Request and Indication Control Data Structures** for more information on the structure.

For a positive response, (**req->resp_err = CNF_RESP_OK**), the request control structure member **resp_info_ptr** is a pointer to the operation-specific data structure (**GETVLIST_RESP_INFO**) for the GetNamedVariableListAttributes function.

For a negative response, or other errors, see **Volume 3 — Module 11 — MMS-EASE Error Handling** for more information.

Return Value: `ST_VOID` (ignored)

Operation-Specific Data Structure Used: **GETVLIST_RESP_INFO**

See page 2-233 for a detailed description of this structure.

Virtual Machine GetNamedVariableListAttributes Operations

The following section contains information on how to use the virtual machine interface for the GetNamed-VariableListAttributes service.

- The GetNamedVariableListAttributes service consists only of the virtual machine function of **`mv_getvlist_resp`**.

There are no virtual machine data structures regarding this service.

Virtual Machine Interface Function

mv_getvlist_resp

Usage: This virtual machine response function allows a user to search through the MMS-EASE database and automatically send a GetNamedVariableListAttributes positive response PDU. This function will return an error if the VMD object is undefined. BE SURE TO SET **`mms_chan_info.objs.vmd`** TO POINT TO THE PROPER VMD STRUCTURE BEFORE CALLING THIS FUNCTION. See **Volume 1 — Module 2 — Channel Information Structure** and **Module 4 — VMD Control** starting on page 2-10 for more information.

Function Prototype: `ST_RET mv_getvlist_resp (MMSREQ_IND *ind);`

Parameters:

`ind` This pointer to the request control data structure of type **`MMSREQ_IND`** is received in the indication service function, **`u_getvlist_ind`**.

Return Value: `ST_RET` **`SD_SUCCESS`**. No Error.

`<> 0` Error Code.

Corresponding User Indication Function: **`u_getvlist_ind`**

12. DeleteNamedVariableList Service

This service is used by a Client application to request that a Server VMD delete one or more Named Variables List objects at a VMD. These must have a MMS Deletable attribute equal to true.

Primitive Level DeleteNamedVariableList Operations

The following section contains information on how to use the paired primitive interface for the DeleteNamedVariableList service. This section covers available data structures used by the PPI, and the four primitive level functions that together make up the DeleteNamedVariableList service. See **Volume 1 — Module 2 — General Sequence of Events — Confirmed Services** for a detailed explanation of the sequence of events that occur within the MMS-EASE environment during confirmed services.

- The DeleteNamedVariableList service consists of the paired primitive functions of **`mp_delvlist`**, **`u_delvlist_ind`**, **`mp_delvlist_resp`**, and **`u_mp_delvlist_conf`**.

Data Structures

Request/Indication

The operation-specific data structure described below is used by the Client in issuing a DeleteNamedVariableList request (**`mp_delvlist`**). It is received by the Server when a DeleteNamedVariableList indication (**`u_delvlist_ind`**) is received.

```
struct delvlist_req_info
  {
  ST_INT16    scope;
  ST_BOOLEAN  dname_pres;
  ST_CHAR     dname [MAX_IDENT_LEN+1];
  ST_BOOLEAN  vnames_pres;
  ST_INT      num_of_vnames;
/*OBJECT_NAME vname_list [num_of_vnames];                          */
  SD_END_STRUCT
  };
typedef struct delvlist_req_info DELVLIST_REQ_INFO;
```

Fields:

`scope` This specifies the scope of the named variable definition(s) to be deleted:

`DELVL_SPEC`. Delete only those variables whose names are in **`vname_list`**.

`DELVL_AA`. The Named Variable List objects are specific to this association (aa-specific). Delete all AA-specific Named Variable List objects.

`DELVL_DOM`. Delete all domain-specific Named Variable List objects in the specified domain (**`dname`**).

`DELVL_VMD`. Delete all VMD-Specific Named Variable List objects.

`dname_pres` **`SD_FALSE`**. Do Not include **`dname`** in the PDU.

`SD_TRUE`. Include **`dname`** in the PDU.

`dname` This contains the name of the domain for which all domain specific variables are to be deleted. Use if **`scope = DELVL_DOM`**.

vnames_pres — **SD_FALSE**. Do Not include **vname_list** in the PDU.

SD_TRUE. Include **vname_list** in the PDU.

num_of_vnames — This indicates the number of variables to be deleted.

vname_list — This array of structures of type **OBJECT_NAME** specifies the specific variables to be deleted. See the description in **Volume 1 — Module 2 — MMS Object Name Structure** for more information on this structure.

NOTE:

1. FOR REQUEST ONLY, when allocating a data structure of type **DELVLIST_REQ_INFO**, enough memory must be allocated to hold the information for the **vnames_list** member of the structure. The following C statement can be used:

```
info = (DELVLIST_REQ_INFO *) chk_malloc (sizeof (DELVLIST_REQ_INFO) +
             (num_of_vnames * (sizeof (OBJECT_NAME))));
```

Response/Confirm

The operation-specific data structure described below is used by the Server in issuing a DeleteNamedVariableList response (**mp_delvlist_resp**). It is received by the Client when a DeleteNamedVariableList confirm (**u_mp_delvlist_conf**) is received.

```
struct delvlist_resp_info
  {
  ST_UINT32 num_matched;
  ST_UINT32 num_deleted;
  };
typedef struct delvlist_resp_info DELVLIST_RESP_INFO;
```

Fields:

num_matched — This indicates the number of named variable list descriptions specified in the request that matched an existing variable.

num_deleted — This indicates the number of named variable lists actually deleted.

Paired Primitive Interface Functions

mp_delvlist

Usage: This primitive request function is used to send a DeleteNamedVariableList request PDU. It uses the data from a structure of type **DELVLIST_REQ_INFO**, pointed to by **info**. This service is used to delete one or more Named Variable Lists previously defined using a DefineNamedVariableList service request.

Function Prototype:

```
MMSREQ_PEND *mp_delvlist (ST_INT chan,
                          DELVLIST_REQ_INFO *info);;
```

Parameters:

chan	This is the channel number over which the request PDU is to be sent.
info	This pointer to structure of type **DELVLIST_REQ_INFO** contains information specific to this request.

Return Value:

MMSREQ_PEND *	This pointer to the request control data structure of type **MMSREQ_PEND** is used to send the request PDU. In case of an error, the pointer is set to null and **mms_op_err** is written with the error code.

Corresponding User Confirmation Function: **u_mp_delvlist_conf**

Operation-Specific Data Structure Used: **DELVLIST_REQ_INFO**

See page 2-241 for a detailed description of this structure.

u_delvlist_ind

Usage: This user function is called when a DeleteNamedVariableList indication is received by a Server node. The contents and operation of this function are user-defined, and depend on the application requirements. To conform to the MMS protocol requirements, the application must do one of the following some time after the indication is received:

1) send a positive response using the primitive response function (**mp_delvlist_resp**) after the specified Variable List has been deleted, or

2) call the appropriate primitive negative (error) response function (**mp_err_resp**). See **Volume 3 — Module 11 — MMS-EASE Error Handling** for an explanation of this function.

See **Volume 1 — Module 1 — User Indication Function Class** for additional information regarding the handling of indications.

Function Prototype:

```
ST_VOID u_delvlist_ind (MMSREQ_IND *ind);
```

Parameters:

`ind` This pointer to the **MMSREQ_IND** structure is described in **Volume 1 — Module 2 — Request and Indication Control Data Structures**. This pointer must be used to send the response to this indication. The primary item of interest in this structure is **ind->req_info_ptr**. This is a pointer to the operation-specific data structure (**DELVLIST_REQ_INFO**) for a DeleteNamedVariableList indication. This pointer will always be valid when **u_delvlist_ind** is called.

Return Value: `ST_VOID` (ignored)

Operation-Specific Data Structure Used: **DELVLIST_REQ_INFO**

See page 2-241 for a detailed description of this structure.

mp_delvlist_resp

Usage: This primitive response function sends a DeleteNamedVariableList positive response PDU. This should be called as a response to the **u_delvlist_ind** function being called (a DeleteNamedVariableList indication is received), and after the specified named variable list was successfully deleted. You should only delete Named Variable List objects that have the MMS deletable attribute set. If you find that a request to delete more than one Named Variable List object is made and at least one of the Named Variable List objects is not MMS deletable while the others were deleted, a positive response should still be sent indicating how many Named Variable List objects were deleted. If you find that you could not delete any of the specified Named Variable List objects (for any reason other than a conflict with the MMS deletable attribute), an error response should be sent using **mp_err_resp** and the additional error information specifying how many of the Named Variable List objects were actually deleted.

Function Prototype:

```
ST_RET mp_delvlist_resp (MMSREQ_IND *ind,           DELV-
                                    LIST_RESP_INFO *info);
```

Parameters:

`ind` This pointer to the indication control structure of type **MMSREQ_IND** is received in the **u_delvlist_ind** function.

`info` This pointer to an Operation-Specific data structure of type **DELVLIST_RESP_INFO** contains information specific to the PDU to be sent.

Return Value: `ST_RET` **SD_SUCCESS**. No Error.

`<> 0` Error Code.

Corresponding User Indication Function: **u_delvlist_ind**

Operation-Specific Data Structure Used: **DELVLIST_RESP_INFO**

See page 2-242 for a detailed description of this structure.

u_mp_delvlist_conf

Usage: This primitive user confirm function is called when the confirm to a DeleteNamedVariableList request (**mp_delvlist**) is received. **resp_err** variable contains a value indicating whether an error occurred. **resp_info_ptr** may contain data that the remote node sent back in the confirmation or it may contain error information if an error occurred. If no error has occurred (**req->resp_err = CNF_RESP_OK**), delete named variable list information is available to the application using the **req->resp_info_ptr** pointer. See **Volume 1 — Module 1 — Primitive User Confirmation Function Class** for specific information on the recommended handling of confirms.

Note that the function called for this event may be changed if desired; however, **u_mp_delvlist_conf** is the standard default function.

Function Prototype:

```
ST_VOID u_mp_delvlist_conf (MMSREQ_PEND *req);
```

Parameters:

`req` This pointer to the **MMSREQ_PEND** structure is returned from the original **mp_delvlist** request function. See **Volume 1 — Module 2 — Request and Indication Control Data Structures** for more information on this structure.

For a positive response (**req->resp_err = CNF_RESP_OK**), the request control structure member **resp_info_ptr** is a pointer to the operation-specific data structure (**DELVLIST_RESP_INFO**) for the DeleteNamedVariableList function.

For a negative response, or other errors, see **Volume 3 — Module 11 — MMS-EASE Error Handling** for more information.

Return Value: `ST_VOID` (ignored)

Operation-Specific Data Structure Used: **DELVLIST_RESP_INFO**

See page 2-242 for a detailed description of this structure.

Virtual Machine DeleteNamedVariableList Operations

The following section contains information on how to use the virtual machine interface for the DeleteVariableAccess service.

- The DeleteNamedVariableList service consists of the virtual machine function of **`mv_delvlist_resp`**.

There are no virtual machine data structures concerning this service.

Virtual Machine Interface Function

mv_delvlist_resp

Usage: This virtual machine response function attempts to delete the specified named variable list(s) from the local Virtual Machine Database. It also responds to a DeleteNamedVariableList indication (**`u_delvlist_ind`**). This function automatically returns an error if there is a problem in deleting a named variable list (for other than a not-Deletable attribute). BE SURE TO SET **`mms_chan_info.objs.vmd`** TO POINT TO THE PROPER VMD STRUCTURE BEFORE CALLING THIS FUNCTION. See **Volume 1 — Module 2 — Channel Information Structure** and **Module 4 — VMD Control** starting on page 2-10 for more information.

Function Prototype: `ST_RET mv_delvlist_resp (MMSREQ_IND *ind);`

Parameters:

`ind` This pointer to the structure of type **`MMSREQ_IND`** is received for indication service function, **`u_delvlist_ind`**.

Return Value: `ST_RET` **`SD_SUCCESS`**. No Error.

`<> 0` Error Code.

Corresponding User Indication Function: **`u_delvlist_ind`**

13. DefineNamedType Service

This service is used by a Client application to request that a Server VMD store a Named Type specification. This is to be used in other definitions of Named Variables or Named Types at the VMD, or accesses to unnamed variables at the VMD.

Primitive Level DefineNamedType Operations

The following section contains information on how to use the paired primitive interface for the DefineNamedType service. This section covers available data structures used by the PPI, and the four primitive level functions that together make up the DefineNamedType service. See **Volume 1 — Module 2 — General Sequence of Events — Confirmed Services** for a detailed explanation of the sequence of events that occur within the MMS-EASE environment during confirmed services.

- The DefineNamedType service consists of the paired primitive functions of **`mp_deftype`**, **`u_deftype_ind`**, **`mp_deftype_resp`**, and **`u_mp_deftype_conf`**.

Data Structures

Request/Indication

The operation-specific data structure described below is used by the Client in issuing a DefineNamedType request (**`mp_deftype`**). It is received by the Server when a DefineNamedType indication (**`u_deftype_ind`**) is received.

```
struct deftype_req_info
  {
  OBJECT_NAME   type_name;
  VAR_ACC_TSPEC type_spec;
  };
typedef struct deftype_req_info  DEFTYPE_REQ_INFO;
```

Fields:

`type_name` This structure of type **`OBJECT_NAME`** contains the name of the Named Type to be defined.

`type_spec` This structure of type **`OBJECT_NAME`** indicates the Type Specification for the Named Type to be defined.

NOTE: See the description in **Volume 1 — Module 2 — MMS Object Name Structure** for information on the **`OBJECT_NAME`** structure, and page 2-150 for information on the **`VAR_ACC_TSPEC`** structure. Also refer to **Volume 1 — Appendix I** for more information on how to build a type specification.

Paired Primitive Interface Functions

mp_deftype

Usage: This primitive request function sends a DefineNamedType request PDU using the data from a structure of type **DEFTYPE_REQ_INFO**, pointed to by **info**. This service is used to store a named type specification for use in the subsequent definition or access of a named variable at the remote node.

Function Prototype:
```
MMSREQ_PEND *mp_deftype (ST_INT chan,
                         DEFTYPE_REQ_INFO *info);
```

Parameters:

chan	This is the channel number over which the request PDU is to be sent.
info	This pointer to a structure of type **DEFTYPE_REQ_INFO** contains information specific to this request.

Return Value:

MMSREQ_PEND *	This pointer to the request control data structure of type **MMSREQ_PEND** is used to send the request PDU. In case of an error, the pointer is set to null and **mms_op_err** is written with the error code.

Corresponding User Confirmation Function: **u_mp_deftype_conf**

Operation-Specific Data Structure Used: **DEFTYPE_REQ_INFO**

See page 2-249 for a detailed description of this structure.

u_deftype_ind

Usage: This user function is called when a DefineNamedType indication is received by a Server node. The contents and operation of this function are user-defined, and depend on the application requirements. To conform to the MMS protocol requirements, the application must do one of the following some time after the indication is received:

1) send a positive response using either:

   a) the primitive response function (**`mp_deftype_resp`**) after the Named Type Specification has been created,

   b) the virtual machine response function (**`mv_deftype_resp`**) to have the virtual machine take the requested action and send the response, or

2) call the appropriate primitive negative (error) response function (**`mp_err_resp`**). See refer to **Volume 3 — Module 11 — MMS-EASE Error Handling** for an explanation of this function.

See **Volume 1 — Module 1 — User Indication Function Class** for additional information regarding the handling of indications.

Function Prototype: `ST_VOID u_deftype_ind (MMSREQ_IND *ind);`

Parameters:

`ind` This pointer to the **`MMSREQ_IND`** structure is described in **Volume 1 — Module 2 — Request and Indication Control Data Structures**. This pointer must be used to send the response to this indication. The primary item of interest in this structure is **`ind->req_info_ptr`**. This is a pointer to the operation-specific data structure (**`DEFTYPE_REQ_INFO`**) for a DefineNamedType indication. This pointer will always be valid when **`u_deftype_ind`** is called.

Return Value: `ST_VOID` (ignored)

Operation-Specific Data Structure Used: **`DEFTYPE_REQ_INFO`**

See page 2-249 for a detailed description of this structure.

mp_deftype_resp

Usage: This primitive response function sends a DefineNamedType positive response PDU. This function should be called as a response to the **u_deftype_ind** function being called (a DefineNamedType indication is received), and after the specified type was successfully created.

Function Prototype: `ST_RET mp_deftype_resp (MMSREQ_IND *ind);`

Parameters:

`ind` This pointer to the indication control structure of type **MMSREQ_IND** is received in the **u_deftype_ind** indication service function.

Return Value: `ST_RET` **SD_SUCCESS**. No Error.

`<> 0` Error Code.

Corresponding User Indication Function: **u_deftype_ind**

Operation-Specific Data Structure Used: NONE

u_mp_deftype_conf

Usage: This primitive user confirm function is called when the confirm to a DefineNamedType request (**mp_deftype**) is received. **resp_err** contains a value indicating whether an error occurred. **resp_info_ptr** may contain data that the remote node sent back in the confirmation or it may contain error information if an error occurred. If no error has occurred (**req->resp_err = CNF_RESP_OK**), DefineNamedType information is available to the application using the **req->resp_info_ptr** pointer. See **Volume 1 — Module 1 — Primitive User Confirmation Function Class** for specific information on the recommended handling of confirms.

Note that the function called for this event may be changed if desired; however, **u_mp_deftype_conf** is the standard default function.

Function Prototype:

```
ST_VOID u_mp_deftype_conf (MMSREQ_PEND *req);
```

Parameters:

`req` This pointer to the **MMSREQ_PEND** structure is returned from the original **mp_deftype** request function. See **Volume 1 — Module 2 — Request and Indication Control Data Structures** for more information on the structure.

For a positive response (**req->resp_err = CNF_RESP_OK**), the request control structure member **resp_info_pres** indicates that there is no confirm data for this function.

For a negative response, or other errors, see **Volume 3 — Module 11 — MMS-EASE Error Handling** for more information.

Return Value: `ST_VOID` (ignored)

Operation-Specific Data Structure Used: NONE

Virtual Machine DefineNamedType Operations

The following section contains information on how to use the virtual machine interface for the DefineNamedType service. It covers the response function that makes up the VMI DefineNamedType service.

- The DefineNamedType service consists of the virtual machine response function of **`mv_deftype_resp`**.

There are no virtual machine data structures concerning this service.

Virtual Machine Interface Function

mv_deftype_resp

Usage: This virtual machine response function attempts to add the specified type to the local Virtual Machine Database of named types. It also responds to a DefineNamedType indication (**`u_deftype_ind`**). This function automatically returns an error if there is a problem adding the type. BE SURE TO SET **`mms_chan_info.objs.vmd`** TO POINT TO THE PROPER VMD STRUCTURE BEFORE CALLING THIS FUNCTION. See **Volume 1 — Module 2 — Channel Information Structure** and **Module 4 — VMD Control** starting on page 2-10 for more information.

Function Prototype: `ST_RET mv_deftype_resp (MMSREQ_IND *ind);`

Parameters:

`ind` This pointer to the structure of type **`MMSREQ_IND`** is received for indication service function, **`u_deftype_ind`**.

Return Value: `ST_RET` **`SD_SUCCESS`**. No Error.

`<> 0` Error Code.

Corresponding User Indication Function: **`u_deftype_ind`**

14. GetNamedTypeAttributes Service

This service is used by the Client application to request that a Server VMD return the attributes of a Named Type object.

Primitive Level GetNamedTypeAttributes Operations

The following section contains information on how to use the paired primitive interface for the GetNamed TypeAttributes service. This section covers available data structures used by the PPI, and the four primitive level functions that together make up the GetNamedTypeAttributes service. See **Volume 1 — Module 2 — General Sequence of Events — Confirmed Services** for a detailed explanation of the sequence of events that occur within the MMS-EASE environment during confirmed services.

- The GetNamedTypeAttributes service consists of the paired primitive functions of **`mp_gettype`**, **`u_gettype_ind`**, **`mp_gettype_resp`**, and **`u_mp_gettype_conf`**.

Data Structures

Request/Indication

The operation-specific data structure described below is used by the Client in issuing a GetNamedTypeAttributes request (**`mp_gettype`**). It is received by the Server when a GetNamedTypeAttributes indication (**`u_gettype_ind`**) is received.

```
struct gettype_req_info
  {
  OBJECT_NAME  type_name;
  };
typedef struct gettype_req_info  GETTYPE_REQ_INFO;
```

Fields:

`type_name` This structure of type **`OBJECT_NAME`** contains the name of the Named Type for which the type definition is to be obtained. See the description in **Volume 1 — Module 2 — MMS Object Name Structure** for more information on this structure.

Response/Confirm

The operation-specific data structure described below is used by the Server in issuing a GetNamedTypeAttributes response (**`mp_gettype_resp`**). It is received by the Client when a GetNamedTypeAttributes confirm (**`u_mp_gettype_conf`**) is received.

```
struct gettype_resp_info
  {
  ST_BOOLEAN    mms_deletable;
  VAR_ACC_TSPEC type_spec;
  };
typedef struct gettype_resp_info GETTYPE_RESP_INFO;
```

Fields:

`mms_deletable` **SD_FALSE**. The Type definition is NOT deletable using a MMS service request.

SD_TRUE. The Type definition is deletable over the network via a MMS service request.

`type_spec` This structure of type **VAR_ACC_TSPEC** contains the type definition for the Named Type. See the description of the variable access data structures starting on page 2-150 for more information on this structure.

Paired Primitive Interface Functions

mp_gettype

Usage: This primitive request function sends a GetNamedTypeAttributes request PDU using the data from a structure of type **GETTYPE_REQ_INFO**, pointed to by **info**. This service is used to obtain the attributes of a named type that exists at the remote node.

Function Prototype:

```
MMSREQ_PEND *mp_gettype (ST_INT chan,
                         GETTYPE_REQ_INFO *info);
```

Parameters:

chan	This is the channel number over which the request PDU is to be sent.
info	This pointer to a structure of type **GETTYPE_REQ_INFO** contains information specific to this request.

Return Value:

MMSREQ_PEND *	This pointer to the request control data structure of type **MMSREQ_PEND** is used to send the request PDU. In case of an error, the pointer is set to null and **mms_op_err** is written with the error code.

Corresponding User Confirmation Function: **u_mp_gettype_conf**

Operation-Specific Data Structure Used: **GETTYPE_REQ_INFO**

See page 2-255 for a detailed description on this structure.

u_gettype_ind

Usage: This user function is called when a GetNamedTypeAttributes indication is received by a Server node. The contents and operation of this function are user-defined, and depend on the application requirements. To conform to the MMS protocol requirements, the application must do one of the following some time after the indication is received:

1) send a positive response using either:

   a) the primitive response function(**`mp_gettype_resp`**) after the Named Type's attributes have been obtained,

   b) the virtual machine response function (**`mv_gettype_resp`**) to have the virtual machine take the requested action and send the response, or

2) call the appropriate primitive negative (error) response function (**`mp_err_resp`**). See **Volume 3 — Module 11 — MMS-EASE Error Handling** for an explanation of this function.

See **Volume 1 — Module 1 — User Indication Function Class** for additional information regarding the handling of indications.

Function Prototype: `ST_VOID u_gettype_ind (MMSREQ_IND *ind);`

Parameters:

`ind` This pointer to the **`MMSREQ_IND`** structure is described in **Volume 1 — Module 2 — Request and Indication Control Data Structures**. This pointer must be used to send the response to this indication. The primary item of interest in this structure is **`ind->req_info_ptr`**. This is a pointer to the operation-specific data structure (**`GETTYPE_REQ_INFO`**) for a GetNamedTypeAttributes indication. This pointer will always be valid when **`u_gettype_ind`** is called.

Return Value: `ST_VOID` (ignored)

Operation-Specific Data Structure Used: **`GETTYPE_REQ_INFO`**

See page 2-255 for a detailed description of this structure.

mp_gettype_resp

Usage: This primitive response function sends a GetNamedTypeAttributes positive response PDU. This function should be called after the **u_gettype_ind** function is called (a GetNamedTypeAttributes indication is received), and after the specified type definition information has been successfully obtained.

Function Prototype:

```
ST_RET mp_gettype_resp (MMSREQ_IND *ind,
                        GETTYPE_RESP_INFO *info);
```

Parameters:

`ind` This pointer to the indication control structure of type **MMSREQ_IND** is received in the **u_gettype_ind** function.

`info` This pointer to an Operation-Specific data structure of type **GETTYPE_RESP_INFO** contains information specific to the PDU to be sent.

Return Value: `ST_RET` **SD_SUCCESS**. No Error.

`<> 0` Error Code.

Corresponding User Indication Function: **u_gettype_ind**

Operation-Specific Data Structure Used: **GETTYPE_RESP_INFO**

See page 2-255 for a detailed description of this structure.

u_mp_gettype_conf

Usage: This primitive user confirmation function is called when the confirm to a GetNamedTypeAttributes request (**mp_gettype**) is received. **resp_err** contains a value indicating whether an error occurred. **resp_info_ptr** may contain data that the remote node sent back in the confirmation or it may contain error information if an error occurred. If no error has occurred (**req->resp_err = CNF_RESP_OK**), GetNamedTypeAttributes information is available to the application using the **req->resp_info_ptr** pointer. See **Volume 1 — Module 1 — Primitive User Confirmation Function Class** for specific information on the recommended handling of confirms.

Note that the function called for this event may be changed if desired; however, **u_mp_gettype_conf** is the standard default function.

Function Prototype:

```
ST_VOID u_mp_gettype_conf (MMSREQ_PEND *req);
```

Parameters:

`req` This pointer to the **MMSREQ_PEND** structure is returned from the original **mp_gettype** request function. See **Volume 1 — Module 2 — Request and Indication Control Data Structures** for more information on the structure.

For a positive response (**req->resp_err = CNF_RESP_OK**), the request control structure member **resp_info_ptr** is a pointer to the operation-specific data structure (**GETTYPE_RESP_INFO**) for the GetNamedTypeAttributes function.

For a negative response, or other errors, see **Volume 3 — Module 11 — MMS-EASE Error Handling** for more information.

Return Value: `ST_VOID` (ignored)

Operation-Specific Data Structure Used: **GETTYPE_RESP_INFO**

See page 2-255 for a detailed description of this structure.

Virtual Machine GetNamedTypeAttributes Operations

The following section contains information on how to use the virtual machine interface for the Get Named Type Attributes service. It covers the virtual machine response function that makes up the Get Named Type Attributes service.

- The GetNamedTypeAttributes service consists of the virtual machine response function of **`mp_gettype_resp`**.

There are no virtual machine data structures concerning this service.

Virtual Machine Interface Function

mv_gettype_resp

Usage: This virtual machine response function allows a user to search through the MMS-EASE database and automatically send a GetNamedTypeAttributes positive response PDU. This function will return an error if it cannot find the specified named type object. BE SURE TO SET **`mms_chan_info.objs.vmd`** TO POINT TO THE PROPER VMD STRUCTURE BEFORE CALLING THIS FUNCTION. See **Volume 1 — Module 2 — Channel Information Structure** and **Module 4 — VMD Control** starting on page 2-10 for more information.

Function Prototype:

```
ST_RET mv_gettype_resp (MMSREQ_IND *ind);
```

Parameters:

`ind`	This pointer to the indication control data structure of type **`MMSREQ_IND`** is received for the indication service function, **`u_gettype_ind`**.

Return Value:

`ST_RET`	**`SD_SUCCESS`**.	No Error.
	`<> 0`	Error Code.

Corresponding User Indication Function: **`u_gettype_ind`**

15. DeleteNamedType Service

This service is used by a Client application to request that a Server VMD delete one or more MMS-defined Named Type objects.

Primitive Level DeleteNamedType Operations

The following section contains information on how to use the paired primitive interface for the DeleteNamedType service. This section covers available data structures used by the PPI, and the four primitive level functions that together make up the DeleteNamedType service. See **Volume 1 — Module 2 — General Sequence of Events — Confirmed Services** for a detailed explanation of the sequence of events that occur within the MMS-EASE environment during confirmed services.

- The DeleteNamedType service consists of the paired primitive functions of **`mp_deltype`**, **`u_deltype_ind`**, **`mp_deltype_resp`**, and **`u_mp_deltype_conf`**.

Data Structures

Request/Indication

The operation-specific data structure described below is used by the Client in issuing a DeleteNamedType request (**`mp_deltype`**). It is received by the Server when a DeleteNamedType indication (**`u_deltype_ind`**) is received.

```
struct deltype_req_info
  {
  ST_INT16    scope;
  ST_BOOLEAN  dname_pres;
  ST_CHAR     dname [MAX_IDENT_LEN+1];
  ST_BOOLEAN  tnames_pres;
  ST_INT      num_of_tnames;
/*OBJECT_NAME tname_list [num_of_tnames];                          */
  SD_END_STRUCT
  };
typedef struct deltype_req_info  DELTYPE_REQ_INFO;
```

Fields:

`scope` This specifies the scope of the type definition(s) to be deleted:

`DELTYPE_SPEC`. Delete only those type names in **`tname_list`**.

`DELTYPE_AA`. The Type Name is specific to this association (aa-specific). Delete all AA-specific types.

`DELTYPE_DOM`. Delete all domain-specific type names in the specified domain (**`dname`**).

`DELTYPE_VMD`. Delete all-VMD Specific type names.

`dname_pres` **`SD_FALSE`**. Do Not include **`dname`** in the PDU.

`SD_TRUE`. Include **`dname`** in the PDU.

`dname` This contains the name of the domain for which all domain specific type names are to be deleted. Used only if **`scope = DELTYPE_DOM`**.

`tnames_pres` **`SD_FALSE`**. Do Not include **`tname_list`** in the PDU.

`SD_TRUE`. Include **`tnames`** in the PDU.

`num_of_tnames` This indicates the number of specific type names to be deleted.

`tname_list` This array of structures of type **`OBJECT_NAME`** contains the specific type names to be deleted. See the description in **Volume 1 — Module 2 — MMS Object Name Structure** for more information on this structure.

NOTE:

1. FOR REQUESTS ONLY, when allocating a data structure of type **`DELTYPE_REQ_INFO`**, enough memory must be allocated to hold the information for the **`tnames_list`** member of the structure. The following C statement can be used:

```
info = (DELTYPE_REQ_INFO *) chk_malloc (sizeof (DELTYPE_REQ_INFO) +
             (num_of_tnames * (sizeof (OBJECT_NAME))));
```

Response/Confirm

The operation-specific data structure described below is used by the Server in issuing a DeleteNamedType response (**`mp_deltype_resp`**). It is received by the Client when a DeleteNamedType confirm (**`u_mp_deltype_conf`**) is received.

```
struct deltype_resp_info
  {
  ST_UINT32 num_matched;
  ST_UINT32 num_deleted;
  };
typedef struct deltype_resp_info DELTYPE_RESP_INFO;
```

Fields:

`num_matched` This indicates the number of named typed descriptions specified in the request that matched an existing named type.

`num_deleted` This indicates the number of named types actually deleted.

Paired Primitive Interface Functions

mp_deltype

Usage: This primitive request function sends a DeleteNamedType request PDU using the data from a structure of type **DELTYPE_REQ_INFO**, pointed to by **info**. This service is used to delete named type specifications previously defined via the DefineNamedType service at the remote node.

Function Prototype:

```
MMSREQ_PEND *mp_deltype (ST_INT chan,
                         DELTYPE_REQ_INFO *info);
```

Parameters:

chan	This is the channel number over which the request PDU is to be sent.
info	This pointer to a structure of type **DELTYPE_REQ_INFO** contains information specific to this request.

Return Value:

MMSREQ_PEND *	This pointer to the request control data structure of type **MMSREQ_PEND** is used to send the request PDU. In case of an error, the pointer is set to null and **mms_op_err** is written with the error code.

Corresponding User Confirmation Function: **u_mp_deltype_conf**

Operation-Specific Data Structure Used: **DELTYPE_REQ_INFO**

See page 2-263 for a detailed description of this structure.

u_deltype_ind

Usage: This user function is called when a DeleteNamedType indication is received by a Server node. The contents and operation of this function are user-defined, and depend on the application requirements. To conform to the MMS protocol requirements, the application must do one of the following some time after the indication is received:

1) send a positive response using either:

   a) the primitive response function (**mp_deltype_resp**) after the specified Named Type has been deleted,

   b) the virtual machine response function (**mv_deltype_resp**) to have the virtual machine take the requested action and send the response, or

2) call the appropriate primitive negative (error) response function (**mp_err_resp**). See **Volume 3 — Module 11 — MMS-EASE Error Handling** for an explanation of this function.

See **Volume 1 — Module 1 — User Indication Function Class** for additional information regarding the handling of indications.

Function Prototype:

```
ST_VOID u_deltype_ind (MMSREQ_IND *ind);
```

Parameters:

`ind` This pointer to the **MMSREQ_IND** structure is described in **Volume 1 — Module 2 — Request and Indication Control Data Structures**. This pointer must be used to send the response to this indication. The primary item of interest in this structure is **ind->req_info_ptr**. This is a pointer to the operation-specific data structure (**DELTYPE_REQ_INFO**) for a DeleteNamedType indication. This pointer will always be valid when **u_deltype_ind** is called.

Return Value: `ST_VOID` (ignored)

Operation-Specific Data Structure Used: **DELTYPE_REQ_INFO**

See page 2-263 for a detailed description of this structure.

mp_deltype_resp

Usage: This primitive response function sends a DeleteNamedType positive response PDU. This function should be called as a response to the **u_deltype_ind** function being called (a DeleteNamedType indication is received), and after ALL the specified deletable types were successfully deleted. Only delete types with the MMS deletable attribute set. If you find that a request to delete more than one type is made and at least one of the types is not MMS deletable while the others were deleted, a positive response should still be sent indicating how many types were deleted. If any of the types specified could not be deleted for any reason other than a conflict with the MMS deletable attribute, an error response should be sent using **mp_err_resp**, with the additional error information specifying how many of the types were actually deleted.

Function Prototype:

```
ST_RET mp_deltype_resp (MMSREQ_IND *ind,
                        DELTYPE_RESP_INFO *info);
```

Parameters:

`ind` This pointer to the indication control structure of type **MMSREQ_IND** is received in the indication service function, **u_deltype_ind**.

`info` This pointer to Operation-Specific data structure of type **DELTYPE_RESP_INFO** contains information specific to the PDU to be sent.

Return Value: `ST_RET` **SD_SUCCESS**. No Error.

`<> 0` Error Code.

Corresponding User Indication Function: **u_deltype_ind**

Operation-Specific Data Structure Used: **DELTYPE_RESP_INFO**

See page 2-264 for a detailed description of this structure.

u_mp_deltype_conf

Usage: This primitive user confirmation function is called when a confirm to a DeleteNamedType request (**mp_deltype**) is received. **resp_err** contains a value indicating whether an error occurred. **resp_info_ptr** may contain data that the remote node sent back in the confirmation or it may contain error information if an error occurred. If no error has occurred (**req->resp_err = CNF_RESP_OK**), DeleteNamedType information is available to the application using the **req->resp_info_ptr** pointer. See **Volume 1 — Module 1 — Primitive User Confirmation Function Class** for specific information on the recommended handling of confirms.

Function Prototype:

```
ST_VOID u_mp_deltype_conf (MMSREQ_PEND *req);
```

Parameters:

`req` This pointer to the **MMSREQ_PEND** structure is returned from the original **mp_deltype** request function. See **Volume 1 — Module 2 — Request and Indication Control Data Structures** for more information on the structure.

For a positive response (**req->resp_err = CNF_RESP_OK**), the request control structure member **resp_info_ptr** is a pointer to the operation-specific data structure (**DELTYPE_RESP_INFO**) for the DeleteNamedType function.

For a negative response, or other errors, see **Volume 3 — Module 11 — MMS-EASE Error Handling** for more information.

Return Value: `ST_VOID` (ignored)

Operation-Specific Data Structure Used: **DELTYPE_RESP_INFO**

See page 2-264 for a detailed description of this structure.

Virtual Machine DeleteNamedType Operations

The following section contains information on how to use the virtual machine interface for the DeleteNamedType service. It covers the virtual machine response function that makes up the DeleteNamedType service.

- The DeleteNamedType service consists of the virtual machine response function of **`mv_deltype_resp`**.

There are no virtual machine data structures concerning this service.

Virtual Machine Interface Function

mv_deltype_resp

Usage: This virtual machine response function attempts to delete the specified named type(s) from the Virtual Machine Database of named types. It also responds to a DeleteNamedType indication (**`u_deltype_ind`**). This function automatically returns an error if there is a problem in deleting the type. BE SURE TO SET **`mms_chan_info.objs.vmd`** TO POINT TO THE PROPER VMD STRUCTURE BEFORE CALLING THIS FUNCTION. See **Volume 1 — Module 2 — Channel Information Structure** and **Module 4 — VMD Control** starting on page 2-10 for more information.

Function Prototype:

```
ST_RET mv_deltype_resp (MMSREQ_IND *ind);
```

Parameters:

`ind` This pointer to the indication control data structure of type **`MMSREQ_IND`** is received for the indication service function, **`u_deltype_ind`**.

Return Value: `ST_RET` **`SD_SUCCESS`**. No Error.

`<> 0` Error Code.

Corresponding User Indication Function: **`u_deltype_ind`**

1. ASN.1 External Handling

Several MMS IS services allow for some types of data to be encoded as ASN.1 EXTERNAL fields. These services are:

Domain Management

- DownloadSegment
- UploadSegment

Program Invocation Management

- Start
- GetProgramInvocationAttributes

The operation-specific data structures for these services handle these cases as complete ASN.1 encoded data elements. To assist the user in encoding and decoding these data elements, a data structure and several functions have been added.

ASN.1 External Data Structure

The structure described below is used to store ASN.1 type EXTERNAL data elements for the services listed above.

```
struct extern_info
  {
  ST_BOOLEAN dir_ref_pres;
  MMS_OBJ_ID dir_ref;
  ST_BOOLEAN indir_ref_pres;
  ST_INT32   indir_ref;
  ST_BOOLEAN dv_descr_pres;
  ST_INT     dv_descr_len;
  ST_UCHAR   *dv_descr;
  ST_CHAR    encoding_tag;
  ST_INT     num_bits;
  ST_INT     data_len;
  ST_UCHAR   *data_ptr;
  SD_END_STRUCT
  };
typedef struct extern_info EXTERN_INFO;
```

Fields:

`dir_ref_pres` **`SD_FALSE`**. Do NOT include **`dir_ref`** in the PDU.

`SD_TRUE`. Include **`dir_ref`** in the PDU.

`dir_ref` This structure of type **`MMS_OBJ_ID`** contains the value of direct reference. Please refer to **ISO 8824, Information Processing Systems —** Open Systems Interconnection — Specification of Abstract Syntax Notation One (ASN.1) for more information. See **Volume 1 — Module 2 — MMS Object Identifier** for information on this structure.

`indir_ref_pres` **`SD_FALSE`**. Do NOT include **`indir_ref`** in the PDU.

`SD_TRUE`. Include **`indir_ref`** in the PDU.

`indir_ref` This contains the value of the indirect reference. Please refer to **ISO 8824, Information Processing Systems —** Open Systems Interconnection **—** Specification of Abstract Syntax Notation One (ASN.1) for more information.

`dv_descr_pres` **`SD_FALSE`**. Do NOT include **`dv_descr`** in the PDU.

`SD_TRUE`. Include **`dv_descr`** in the PDU.

`dv_descr_len` This is the length, in bytes, of the data pointed to by **`dv_descr`**.

`dv_descr` This is a pointer to the data value description.

`encoding_tag` This indicates the type of encoding:

- **`0`** single ASN.1
- **`1`** octet aligned
- **`2`** arbitrary

`num_bits` This indicates the number of bits to be sent in the data. Used only if **`encoding_tag==2`**.

`data_len` This is the length, in bytes, of the data pointed to by **`data_ptr`**.

`data_ptr` This is a pointer to the EXTERNAL data.

ASN.1 External Handling Support Functions

These functions are called by the user's application program to manipulate EXTERNAL data elements required by DownloadSegment, UploadSegment, Start and GetProgramInvocation Attributes. The contents of these functions are determined by MMS-EASE.

ms_encode_extern

Usage: This support function is used to create an ASN.1 data element. You need to pass in a pointer to an ASN.1 build buffer used to encode the EXTERNAL data element, and the length of the buffer. In addition, pointers must be provided to output parameters.

Function Prototype:

```
ST_RET ms_encode_extern (EXTERN_INFO *src,
                         ST_UCHAR *asn1dest,
                         ST_INT destlen,
                         ST_INT *len_out,
                         ST_UCHAR **asn1_start_out);
```

Parameters:

`src`	This structure of type **`EXTERN_INFO`** points to the EXTERNAL data element to be encoded. See page 2-271 for a detailed description of this structure.
`asn1dest`	This is a pointer to the ASN.1 build buffer that will contain the EXTERNAL data element.
`destlen`	This contains the actual length of the EXTERNAL data element.
`len_out`	This is the pointer to the maximum length of the ASN.1 build buffer.
`asn1_start_out`	This pointer points to the start of the EXTERNAL data element.

Return Value: `ST_RET`

`SD_SUCCESS`. No Error.

`<> 0` Error Code.

ms_decode_extern

Usage: This support function is used to decode an ASN.1 data element and place the results in a data structure of type **EXTERN_INFO**. You need to pass in a pointer to an ASN.1 EXTERNAL data element, and the length of this element. In addition, a pointer must be provided to the **EXTERN_INFO** structure.

Function Prototype:

```
ST_RET ms_decode_extern (ST_UCHAR *asn1,
                         ST_INT asn1len,
                         EXTERN_INFO *dest);
```

Parameters:

asn1	This is the pointer to the ASN.1 EXTERNAL data element.
asn1len	This is the the maximum length of the ASN.1 EXTERNAL data element.
dest	This structure of type **EXTERN_INFO** points to the destination to place to EXTERNAL data element to be decoded. See page 2-271 for a detailed description of this structure.

Return Value:

ST_RET	**SD_SUCCESS**.	No Error.
	<> 0	Error Code.

2. Domain Management Introduction

MMS provides a set of services that allows domains to be uploaded from the device or downloaded to the device. Domains represent real resources in the server that are available to the client. They may contain specific information such as program instructions, tables, or data, or nothing at all. Domains are also classified as STATIC or DYNAMIC. STATIC domains represent predefined resources within the server. They are known by name and cannot be deleted. DYNAMIC domains may be created and deleted through MMS services.

Domain Management consists of twelve services:

InitiateDownloadSequence This service commands the VMD (Server) to create the domain and prepare to receive a download. See page 2-291 for more information.

DownloadSegment This service is requested by the Server to have a segment of download information transferred by the Client. See page 2-297 for more information.

TerminateDownloadSequence This service is requested by the Server to notify the Client that the download sequence is complete. See page 2-303 for more information.

InitiateUploadSequence This service is used by the Client to request that the Server prepare to upload the contents of a domain. See page 2-315 for more information.

UploadSegment This service is requested by the Client to have a segment of upload information transferred by the Server. See page 2-321 for more information.

TerminateUploadSequence This service is requested by the Client to request that the Server terminate an upload sequence. See page 2-327 for more information.

Request Domain Download This service is used by a VMD (Server) to request that a Client perform a download in the VMD. See page 2-339 for more information.

RequestDomainUpload This service is used by a VMD (Server) to request that the contents of a specified domain be uploaded to the Client. See page 2-345 for more information.

LoadDomainContent This service is used to tell a VMD to download (load) a domain from a file. The file may be local to the VMD or may be contained on an external file server. See page 2-351 for more information.

StoreDomainContent This service is used to tell a VMD to upload (store) a domain from a file. The file may be local to the VMD or may be contained on an external file server. See page 2-357 for more information.

DeleteDomain This service is used by a Client to delete an existing domain, usually before initiating a download sequence. See page 2-363 for more information.

GetDomainAttributes This service is used to obtain the attributes of a specified domain. See page 2-369 for more information.

3. Virtual Machine Interface

The MMS-EASE virtual machine defines several variables, data structures, and functions to assist applications with managing information relating to MMS domains. Per the MMS specification, a domain represents a subset of capabilities of a VMD (Virtual Manufacturing Device). In some VMDs, there might be a domain for the operating system and another domain for the program to be run under the operating system. MMS-EASE supports the ability to model multiple VMDs within a single application process and maintains an alphabetized list of named domains. This list can be manipulated using the appropriate support functions (e.g., **`ms_add_named_dom`**, **`ms_del_named_dom`**), for each VMD modeled. See page 2-282 for more information on these support functions. These VMDs and domains are used by the virtual machine during variable access. See **Module 4 — VMD Management** starting on page 2-10 for more information on VMDs.

The following section describes the data structures used by the MMS-EASE virtual machine to hold the domain management information. These same structures also can be accessed and written by your application programs.

Common Domain Data Structures

Domain Object Structure

The following structure defines MMS objects (such as variables, types, events, semaphores) that may be a part of a domain. This structure is used to hold pointers to the lists of various objects contained within the domain. Do not access any members of this structure marked with **/* DO NOT USE */**.

```
struct domain_objs
  {
  struct named_var       *var_list;
  struct named_var_list  *var_list_list;
  struct named_type      *type_list;
  struct scat_acc         *scat_acc_list;
  struct sem             *sem_list;
  struct event_cond      *ev_cond_list;
  struct event_action    *ev_act_list;
  struct journal          *jour_list;
  ST_INT num_named_var;
  ST_INT max_named_var;
  struct named_var       **named_var_array;
  ST_INT num_nvlist;
  ST_INT max_nvlist;
  struct named_var_list   **nvlist_array;
  ST_CHAR rsrvd[8];
  };
typedef struct domain_objs DOMAIN_OBJS;
```

Fields:

`var_list` This pointer to a list of **`NAMED_VAR`** structures contains definitions of all named variables associated with this domain. See **Module 5 — Variable Access** on page 2-59 for more information on this structure.

`var_list_list`[1] This pointer to a list of **`NAMED_VAR_LIST`** structures contains definitions of all named variable lists associated with this domain. See **Module 5 — Variable Access** on page 2-213 for more information on this structure.

`type_list`[1] This pointer to a list of **`NAMED_TYPE`** structures contains definitions of all named types associated with this domain. See **Module 5 — Variable Access** on page 2-62 for more information on this structure.

`scat_acc_list`[1] This pointer to a list of **`SCAT_ACC`** structures contains definitions of all scattered access variables associated with this domain.

`sem_list`[1] This pointer to a list of type **`SEM`** structures contains definitions of all semaphores associated with this domain. See **Volume 3 — Module 7 — Semaphore Management** for more information on this structure.

`ev_cond_list`[1] This pointer to a list of type **`EVENT_COND`** structures contains definitions of all event conditions associated with this domain. See **Volume 3 — Module 7 — Event Management** for more information on this structure.

`ev_act_list`[1] This pointer to a list of type **`EVENT_ACTION`** structures contains definitions of all event actions associated with this domain. See **Volume 3 — Module 7 — Event Management** for more information on this structure.

`jour_list`[4] This pointer to a list of type **`JOURNAL`** structures contains definitions of all journals associated with this domain. See **Volume 3 — Module 7 — Journal Management** for more information on this structure.

The following elements of this structure are used instead of the **`var_list`** and **`var_list_list`** if the binary search option is used instead of linked lists:

`num_named_var` This indicates the actual number of named variables associated with this domain.

`max_named_var` This indicates the maximum number of named variables associated with this domain.

`named_var_array` This array of pointers points to a list of **`NAMED_VAR`** structures containing definitions of all named variables associated with this domain. See **Module 5 — Variable Access** on page 2-59 for more information on this structure.

`num_named_nvlist` This indicates the actual number of named variable lists associated with this domain.

`max_named_nvlist` This indicates the maximum number of named variable lists associated with this domain.

`nvlist_array` This array of pointers points to a list of **`NAMED_VAR_LIST`** structures containing definitions of all named variable lists associated with this domain. See **Module 5 — Variable Access** on page 2-213 for more information on this structure.

`rsvd` This eight character array is provided for application use. A typical practice for this structure member is to maintain a reference to another data structure in the application that contains information related to the use of the domain. MMS-EASE does not modify the contents of this structure member.

[4] These are currently defined in skeletal form only, and are used as placeholders. The user may expand or modify these structures.

Named Domain Control Structure

The following structure defines a particular named domain. It provides pointers to MMS objects contained within that domain. This structure is typically used as one member of a list of named `domains`. Do not access any members of this structure marked with **/* DO NOT USE */.**

```
struct named_dom_ctrl
  {
  DBL_LNK     link;
  ST_CHAR     dom_name[MAX_IDENT_LEN+1];
  ST_UCHAR    protection;
  ST_BOOLEAN  deletable;
  ST_BOOLEAN  sharable;
  ST_INT      state;
  ST_INT8     upl_in_prog;
  ST_INT32    octet_pos;
  ST_INT      npi;
  ST_CHAR     reserved[4];
  ST_CHAR     user_rsrvd[8];
  ST_CHAR     *dom_content;
  DOMAIN_OBJS objs;
  ST_BOOLEAN  dl_detail_pres;
  ST_INT      dl_detail_len;
  ST_UCHAR    *dl_detail;
  ST_INT      num_of_capab;
/*ST_CHAR     *capab_list [num_of_capab];                          */
  SD_END_STRUCT
  };
typedef struct named_dom_ctrl NAMED_DOM_CTRL;
```

Fields:

`link` /* FOR INTERNAL USE ONLY, DO NOT USE */

`dom_name` This contains the name of the domain. This must be Visible String of no longer than 32 characters existing only of numbers (0-9), upper case letters (A-Z), lower case letters (a-z), and the **_** and **$** characters. It must not start with a number or contain spaces. It is a MMS identifier and is null-terminated.

`protection` This is an access code that can be compared to the privilege allowed for a given remote communications entity so that access to this domain can be controlled. This member is not currently used by MMS-EASE.

`deletable` **SD_TRUE**. This domain can be deleted by the remote application.

SD_FALSE. This domain cannot be deleted by the remote application.[5]

`sharable` **SD_TRUE**. This domain can be referenced by more than one program invocation.

SD_FALSE. This domain can only be referenced by one program invocation.

`state` This value represents the current state of the domain. Possible values for this member are:

DOM_NON_EXISTENT. This indicates that the domain does not exist.

DOM_LOADING. This indicates that the domain is an intermediate state that occurs during the loading process.

DOM_READY. The domain is in a ready state following a successful download.

DOM_IN_USE. This indicates that there are one or more program invocations defined for this domain.

[5] MMS-EASE does not use this member of the **NAMED_DOM_CTRL** structure. If you tell MMS-EASE to delete a named domain using a call to **ms_del_named_dom**, it will delete it. This member is here only for use by the user's application program.

DOM_COMPLETE. This is an intermediate state that occurs after the last DownloadSegment has been received but before the TerminateDownloadSequence has been called.

DOM_INCOMPLETE. This is an intermediate state that occurs when a Download Sequence is terminated before the loading process is completed.

DOM_D1. The rest of these values represent intermediate states (i.e., states between a **DOM_D2**. request and its response).

DOM_D3.
DOM_D4.
DOM_D5.
DOM_D6.
DOM_D7.
DOM_D8.
DOM_D9.

`upl_in_prog` **SD_TRUE**. This domain is currently being uploaded.

SD_FALSE. This domain is not currently being uploaded.

`octet_pos` This contains the number of octets that have been loaded. It is used when the domain is being downloaded.

`npi` This contains a count of the number of program invocations having references to this domain.

`user_rsrvd` The data in this member is user-defined, and can be used for any purpose.

`dom_content` This is a pointer set to null on creation. It is not used by MMS-EASE. It is set aside for application-specific use.

`objs` This structure of type **DOMAIN_OBJS** contains the lists of the named MMS objects attached to this domain. See page 2-207 for a detailed description of this structure.

`dl_detail_pres` **SD_FALSE**. Do NOT include **dl_detail** in the PDU.

SD_TRUE. Include **dl_detail** in the PDU.

`dl_detail_len` This is the length, in bytes, of the data pointed to by **dl_detail**.

`dl_detail` This is a pointer to the Download Detail. The content of this data must be in the ASN.1 format and must conform to the particular companion standard governing the domain to be downloaded.

`num_of_capab` This indicates the number of character strings pointed to by the elements of **capab_list**.

`capab_list` This array of pointers to implementation specific null-terminated character strings describes the capabilities of the VMD. These are a part of the domain to be downloaded.

NOTE: The MMS-EASE Virtual Machine function automatically allocates dynamic memory. This is to be used for storage of the download detail capability pointers when the structure is created during a call to **ms_add_named_dom**. The user does not need to be concerned with the management of this memory. See page 2-282 for more information.

Application-Association Specific Domain Objects

The list of objects, defined as Application-Association-specific, is pointed to by the **objs.aa_objs** member of the **MMS_CHAN_INFO** structure. Objects of this type exist only as long as the association over which they were created is still active. See **MMS_CHAN_INFO** in **Volume 1 - Module 2** and **DOMAIN_OBJS** on page 2-277 for more information.

NOTE: When a channel is concluded or aborted, the domain objects defined for that channel will have to be deleted using a call to **ms_del_domain_objs**.

Domain Variables

```
extern ST_INT mms_dom_count;
```

This variable keeps a running count of all the named domains currently defined for all the defined VMDs.

```
extern ST_INT max_mmsease_doms;
```

This variable indicates the maximum number of domains that can be created by the VMI. The default value is 20.

Domain Support Functions

These functions are called by the user's application program to perform the various support functions required by MMS-EASE to manage domains. For Domain Management, they are used to add and delete domains and domain objects. The contents of these functions are determined by MMS-EASE.

ms_add_named_dom

Usage: This support function allocates memory for a named domain control structure (**NAMED_DOM_CTRL**) and inserts it into the alphabetized list of named domains. The first element in the list is pointed to by the **dom_list** member of the **VMD_CTRL** structure for the VMD. This will contain this domain. In this appropriate **NAMED_DOM_CTRL** structure, this function also fills in the following members with the specified defaults:

dom_name	name of the domain
upl_in_prog	**SD_FALSE**
octet_pos	**0**
npi	**0**
state	**DOM_LOADING**

After this function executes properly and returns, other variables can be manipulated in this structure, as necessary and appropriate. After the domain is loaded, the state should be set to **DOM_READY** or **DOM_IN_USE** as appropriate. Be sure to set **m_vmd_select** to point to the appropriate VMD control structure that contains this domain. See **Module 4 — VMD Support** starting on page 2-10 for more information on VMDs.

Function Prototype:

```
NAMED_DOM_CTRL *ms_add_named_dom (ST_CHAR *name,
                                  ST_UCHAR protection);
```

Parameters:

`name` This is a pointer to the name of the domain to be created.

`protection` This value passed in this parameter is placed in the **protection** member of the **NAMED_DOM_CTRL** structure corresponding to the domain that was added.

Return Value:

`NAMED_DOM_CTRL *` This pointer to the structure of type **NAMED_DOM_CTRL** is assigned to hold the information about this named domain. In case of an error, the pointer is set to null and **mms_op_err** is written with the error code.

ms_add_named_domain

Usage: This support function allocates memory for a named domain control structure (**NAMED_DOM_CTRL**) and inserts it into the alphabetized list of named domains. The first element in the list is pointed to by the **dom_list** member of the **VMD_CTRL** structure for the VMD. This will contain this domain. The **m_vmd_select** pointer must be set to point to the appropriate **VMD_CTRL** structure. See **Module 4 — VMD Support** starting on page 2-10 for more information on VMDs.

This function is very similar to **ms_add_named_dom**, in that it assigns default values to the following members of the **NAMED_DOM_CTRL** structure:

dom_name	name of the domain
upl_in_prog	**SD_FALSE**
octet_pos	**0**
npi	**0**
state	**DOM_LOADING**

however, it has two distinct advantages:

1) it creates and initializes the domain control structure without requiring the user to "unpack" the data values from an InitiateDownloadSequence indication, and passes them to **ms_add_named_dom**, and

2) it automatically allocates memory for, and attaches, the domain-specific capability lists.

After the domain has been loaded, the state should be set to **DOM_READY** or **DOM_IN_USE**.

Function Prototype:
```
NAMED_DOM_CTRL *ms_add_named_domain(INITDOWN_REQ_INFO *dom,
                                   ST_UCHAR protection);
```

Parameters:

`dom` This pointer to a structure contains the information necessary to fill in the **NAMED_DOM_CTRL** structure. When in the context of the **u_initdown_ind** function, MMS-EASE supplies this pointer by referencing the **req_info_ptr** of the **MMSREQ_PEND**.

`protection` This value passed in this parameter is placed in the **protection** member of the **NAMED_DOM_CTRL** structure for the created domain.

Return Value:

`NAMED_DOM_CTRL *` This pointer to the structure of type **NAMED_DOM_CTRL** contains the data about this domain. See page 2-279 for more information. In case of an error, the pointer is set to null and **mms_op_err** is written with the error code.

ms_del_all_named_doms

Usage: This support function deletes an entire list of **NAMED_DOM_CTRL** structures. Any domain objects, contained in the scope of these named domains, are deleted as well.

Function Prototype:

```
ST_VOID ms_del_all_named_doms (NAMED_DOM_CTRL *dom_list);
```

Parameters:

dom_list	This is a pointer to the first **NAMED_DOM_CTRL** structure in the list of named domains.

Return Value: ST_VOID (ignored)

ms_del_domain_objs

Usage: This support function deletes all the domain objects corresponding to the **DOMAIN_OBJS** structure passed to this function. This function can be useful for deleting all Application-Association (AA)-specific objects, pointed to by **mms_chan_info[chan].objs.aa_objs**, when terminating associations. It also can be useful when you wish to delete all a domain's objects without deleting the domain. Be sure to set **m_vmd_select** to point to the appropriate VMD control structure containing this domain objects structure. This applies if the domain objects are part of a named domain. See **Volume 1 — Module 2 — MMS_CHAN_INFO** and **Module 4 — VMD Support** starting on page 2-10 for more information.

Function Prototype:

```
ST_VOID ms_del_domain_objs (DOMAIN_OBJS *objs);
```

Parameters:

objs	This is a pointer to the domain object structure, **DOMAIN_OBJS**, for all the objects to be deleted.

Return Value: ST_VOID (ignored)

ms_del_named_dom

Usage: This support function deletes a named domain from the **NAMED_DOM_CTRL** structure list. All domain objects (such as variables, semaphores) are also deleted, as necessary. All memory allocated specifically for this domain will be freed. Types belonging to this domain are marked so that any new variables will not be associated with these types. The **deletable** element of the domain is ignored. Any defined program invocations must be modified or deleted that referred to the deleted domain. Be sure to set **m_vmd_select** to point to the appropriate VMD control structure that contains this domain. See **Module 4 — VMD Support** starting on page 2-10 for more information on VMDs.

Function Prototype: `ST_RET ms_del_named_dom (ST_CHAR *dom);`

Parameters:

`dom` This is a pointer to the name of the domain to be deleted.

Return Value: `ST_RET` **SD_SUCCESS**. No Error.

`<> 0` Error Code.

ms_find_dom_objs

Usage: This support function obtains a pointer to a structure, of type **DOMAIN_OBJS** containing pointers to a list of various types of objects for the scope of the object specified. After calling this function, the pointer provided can be used to look for a specific variable or type. This is accomplished by calling a support function such as **ms_find_named_var**, or **ms_find_named_type**. Be sure to set **m_vmd_select** to point to the appropriate VMD control structure containing this domain if the object is not association specific. See **Module 4 — VMD Support** starting on page 2-10 for more information on VMDs.

Function Prototype:

```
struct domain_objs *ms_find_dom_objs (OBJECT_NAME *obj,
                                      ST_INT chan);
```

Parameters:

obj	This pointer to a structure of type **OBJECT_NAME** specifies the object for which to search. See **Volume 1 — Module 2 — MMS Object Name Structure** for more information on this structure.
chan	This is the channel number used when the object being searched for is Application-Association (AA)-specific.

Return Value:

DOMAIN_OBJS	*	This pointer to the structure of type **DOMAIN_OBJS** contains a reference to the specified object. In case of an error, the pointer is set to null and **mms_op_err** is written with the error code.

ms_find_named_dom

Usage: This support function obtains a pointer to the **NAMED_DOM_CTRL** structure corresponding to the specified named domain. Be sure to set **m_vmd_select** to point to the appropriate VMD control structure containing this domain. See **Module 4 — VMD Support** starting on page 2-10 for more information on VMDs.

Function Prototype:

```
NAMED_DOM_CTRL *ms_find_named_dom (ST_CHAR *name);
```

Parameters:

name	This is a pointer to the name of the domain for which the **NAMED_DOM_CTRL** structure is to be found.

Return Value:

NAMED_DOM_CTRL *	This pointer to the structure of type **NAMED_DOM_CTRL** corresponds to the specified domain. In case of an error, the pointer is set to null and **mms_op_err** is written with the error code.

ms_set_mv_download_sharable

Usage: This support function is used to change the value if the **sharable** parameter sent in the Initiate Download Request as part of calling the **mv_forced_download** function. The default value of the **sharable** parameter is **SD_TRUE**. Once the value is changed using this function the new value will continue to be the **sharable** value sent in the InitiateDownloadRequest brought about in the **mv_forced_download** state machine. See page 2-314 for more information on **mv_forced_download** service.

Functional Prototype: `ST_VOID ms_set_mv_download_sharable (ST_BOOLEAN val);`

Parameters:

`val`	**SD_TRUE**. Download **sharable** attribute is set to true.
	SD_FALSE. Download **sharable** attribute is set to false.

Return Value: `ST_VOID` (ignored)

4. InitiateDownloadSequence Service

This service is used to request that the server create the named Domain and begin its loading.

Primitive Level InitiateDownloadSequence Operations

The following section contains information on how to use the paired primitive interface for the InitiateDownloadSequence service. It covers available data structures used by the PPI, and the four primitive level functions that together make up the InitiateDownloadSequence service. See **Volume 1 — Module 2 — General Sequence of Events — Confirmed Services** for a detailed explanation of the sequence of events that occur within the MMS-EASE environment during confirmed services.

- The InitiateDownloadSequence service consists of the paired primitive functions of **`mp_initdown`**, **`u_initdown_ind`**, **`mp_initdown_resp`**, and **`u_mp_initdown_conf`**.

Data Structures

Request/Indication

The operation specific structure described below is used by the Client in issuing an InitiateDownloadSequence request (**`mp_initdown`**). It is received by the Server when an InitiateDownloadSequence indication (**`u_initdown_ind`**) is received.

```
struct initdown_req_info
  {
  ST_CHAR    dname [MAX_IDENT_LEN +1];
  ST_BOOLEAN sharable;
  ST_INT     num_of_capab;
/*ST_CHAR *  capab_list [num_of_capab];                    */
  SD_END_STRUCT
  };
typedef struct initdown_req_info INITDOWN_REQ_INFO;
```

Field:

`dname` This contains the name of the domain to be downloaded.

`sharable` **`SD_FALSE`**. After loading the specified domain, it may be used by more than one Program Invocation concurrently.

`SD_TRUE`. After loading the specified domain, it may only be used by one Program Invocation at a time.

`num_of_capab` This indicates the number of pointers in the capabilities list pointed to by **`capab_list`**.

`capab_list` This pointer to an implementation-specific character string describes the capabilities and limitations on the resources of the VMD that are a part of the domain to be loaded.

NOTE: FOR REQUEST ONLY, when allocating a data structure of type **`INITDOWN_REQ_INFO`**, enough memory must be allocated to hold the information for the capabilities list contained in **`capab_list`**. The following C statement can be used:

```
info = (INITDOWN_REQ_INFO *) chk_malloc(sizeof(INITDOWN_REQ_INFO) + (num_capab *
            sizeof(ST_CHAR *)));
```

Paired Primitive Interface Functions

mp_initdown

Usage: This primitive request function sends an InitiateDownloadSequence request PDU, using the data from a structure of type **INITDOWN_REQ_INFO**, pointed to by **info**. This service is used to signal the beginning of a download sequence to the remote node. The act of initiating a download to a domain will cause its creation if it did not already exist.

Function Prototype:

```
MMSREQ_PEND *mp_initdown (ST_INT chan,
                          INITDOWN_REQ_INFO *info);
```

Parameters:

chan	This integer contains the channel number over which the Initiate Download Sequence PDU is to be sent.
info	This pointer to a structure of type **INITDOWN_REQ_INFO** contains information specific to the request PDU to be sent.

Return Value:

MMSREQ_PEND *	This pointer to the request control data structure of type **MMSREQ_PEND** is used to send the InitiateDownloadSequence PDU. In case of an error, the pointer is set to null and **mms_op_err** is written with the error code.

Corresponding User Confirmation Function: **u_mp_initdown_conf**

Operation-Specific Data Structure Used: **INITDOWN_REQ_INFO**

See page 2-291 for a detailed description of this structure.

u_initdown_ind

Usage: This user indication function is called when an InitiateDownloadSequence indication is received by a Server node. The contents and operation of this function are user-defined and depend on the application requirements. To conform to the MMS protocol requirements, the application must do one of the following some time after the indication is received:

1) send a positive response using the primitive response function (**mp_initdown_resp**) after successfully preparing to download the specified domain, or

2) call the appropriate primitive negative (error) response function (**mp_err_resp**). See **Volume 3 — Module 11 — MMS-EASE Error Handling** for more information on this function.

See **Volume 1 — Module 1 — User Indication Function Class** for additional information on the recommended handling of indications.

Function Prototype:

```
ST_VOID u_initdown_ind (MMSREQ_IND *ind);
```

Parameters:

`ind` This pointer to the **MMSREQ_IND** structure is described in **Volume 1 — Module 2 —Request and Indication Control Data Structures**. This pointer must be used to send the response to this indication. The primary item of interest in this structure is **ind->req_info_ptr**. This is a pointer to the operation specific data structure for an InitiateDownloadSequence indication (**INITDOWN_REQ_INFO**). This pointer will always be valid when **u_initdown_ind** is called.

Return Value: ST_VOID (ignored)

Operation-Specific Data Structure Used: **INITDOWN_REQ_INFO**

See page 2-291 for a detailed description of this structure.

mp_initdown_resp

Usage: This primitive response function sends an InitiateDownloadSequence positive response PDU. This function should be called as a response to the **u_initdown_ind** function being called (an InitiateDownloadSequence indication is received), and after successfully preparing to download the specified domain. Once you have responded positively to the InitiateDownloadSequence indication, you should issue consecutive DownloadSegment requests using **mp_download** to obtain the required download data.

Function Prototype: `ST_RET mp_initdown_resp (MMSREQ_IND *ind);`

Parameters:

`ind` This pointer to the indication control data structure of type **MMSREQ_IND** is passed to the **u_initdown_ind** function when it was called.

Return Value: `ST_RET` **SD_SUCCESS**. No Error.

`<> 0` Error Code.

Corresponding User Indication Function: **u_initdown_ind**

Operation-Specific Data Structure Used: NONE

u_mp_initdown_conf

Usage: This primitive user confirmation function is called when a confirm to an InitiateDownload-Sequence request (**mp_initdown**) is received. **resp_err** contains a value indicating whether an error occurred. **resp_info_ptr** may contain data that the remote node sent back in the confirmation or it may contain error information if an error occurred. If no error has occurred (**req->resp_err = CNF_RESP_OK**), InitiateDownloadSequence information is available to the application using the **req->resp_info_ptr** pointer. See **Volume 1 — Module 1 — Primitive User Confirmation Function Class** for more information on the recommended handling of confirms.

Note that the function called for this event may be changed if desired; however, **u_mp_initdown_conf** is the standard default function.

Function Prototype:

```
ST_VOID u_mp_initdown_conf (MMSREQ_PEND *req);
```

Parameters:

`req` This pointer to the **MMSREQ_PEND** structure is returned from the original **mp_initdown** request. See **Volume 1 — Module 2 — Request and Indication Control Data Structures** for more information on this structure.

For a positive response (**req->resp_err = CNF_RESP_OK**), the request control structure member **resp_info_pres** indicates that there is no confirm data for this function.

For a negative response, or other errors, see **Volume 3 — Module 11 — MMS-EASE Error Handling** - for more information.

Return Value: `ST_VOID` (ignored)

Operation-Specific Data Structures Used: NONE

5. DownloadSegment Service

This service is requested by the Server to have a segment of download information transferred by the Client.

Primitive Level DownloadSegment Operations

The following section contains information on how to use the paired primitive interface for the DownloadSegment service. It covers available data structures used by the PPI, and the four primitive level functions that together make up the DownloadSegment service. See **Volume 1 — Module 2 — General Sequence of Events — Confirmed Services** for a detailed explanation of the sequence of events that occur within the MMS-EASE environment during confirmed services.

- The DownloadSegment service consists of the paired primitive functions of **`mp_download`**, **`u_download_ind`**, **`mp_download_resp`,** and **`u_mp_download_conf`**.

Data Structures

Request/Indication

The operation specific structure described below is used by the client in issuing a DownloadSegment request (**`mp_download`**). It is received by the server when a DownloadSegment indication (**`u_download_ind`**) is received.

```
struct download_req_info
  {
  ST_CHAR dname [MAX_IDENT_LEN +1];
  };
typedef struct download_req_info DOWNLOAD_REQ_INFO;
```

Fields:

`dname` This contains the name of the domain to be downloaded

Response/Confirm

The operation specific data structure described below is used by the Server in issuing a DownloadSegment response (**mp_download_resp**). It is received by the Client when a Download Segment confirm (**u_mp_download_conf**) is received.

```
struct download_resp_info
  {
  ST_INT     load_data_type;
  ST_INT     load_data_len;
  ST_UCHAR   *load_data;
  ST_BOOLEAN more_follows;
  };
typedef struct download_resp_info DOWNLOAD_RESP_INFO;
```

Fields:

load_data_type	This indicates the format for the data to be downloaded: **LOAD_DATA_NON_CODED**. This is an octet string. **LOAD_DATA_CODED**. This is a value encoded by some other external source - ASN.1 EXTERNAL data element. See page 2-271 for more information on how to encode and decode ASN.1 EXTERNAL data elements.
load_data_len	This is the length, in bytes, of the data pointed to by **load_data**.
load_data	This is a pointer to the data that is to be downloaded.
more_follows	**SD_FALSE**. There is NO MORE DATA to be transmitted. **SD_TRUE**. There is more data to be transmitted in order to complete the download. This is the default.

Paired Primitive Interface Functions

mp_download

Usage: This primitive request function sends a DownloadSegment request PDU to a remote node. It uses the data found in a structure of type **DOWNLOAD_REQ_INFO**, pointed to by **info**. The DownloadSegment service is used to tell the remote node to transmit another segment of the domain being downloaded. This request is sent by the node being downloaded, not the node that initiated the download sequence.

Function Prototype:

```
MMSREQ_PEND *mp_download (ST_INT chan,
                          DOWNLOAD_REQ_INFO *info);
```

Parameters:

`chan` This is the channel number over which the request PDU is to be sent.

`info` This pointer to an operation-specific data structure of type **DOWNLOAD_REQ_INFO** contains information specific to the request PDU to be sent.

Return Value:

`MMSREQ_PEND *` This pointer to the request control data structure of type **MMSREQ_PEND** is used to send the PDU. In case of an error, the pointer is set to null and **mms_op_err** is written with the error code.

Corresponding User Confirmation Function: **u_mp_download_conf**

Operation-Specific Data Structure Used: **DOWNLOAD_REQ_INFO**

See page 2-297 for a detailed description of this structure.

u_download_ind

Usage: This user indication function is called when a DownloadSegment indication is received by a Server node. The contents and operation of this function are user-defined and depend on the application requirements. To conform to the MMS protocol requirements, the application must do one of the following some time after the indication is received:

1) send a positive response using the primitive positive response function (**`mp_downoad_resp`**) after the required segment of the download has been obtained, or

2) call the appropriate primitive negative (error) response function (**`mp_err_resp`**). See **Volume 3 — Module 11 — MMS-EASE Error Handling** for more information on this function.

See **Volume 1 — Module 1 — User Indication Function Class** for additional information on the recommended handling of indications.

Function Prototype:

```
ST_VOID u_download_ind (MMSREQ_IND *ind);
```

Parameters:

`ind` This pointer to the **`MMSREQ_IND`** structure is described in **Volume 1 — Module 2 — Request and Indication Control Data Structures**. This pointer must be used to send the response to this indication. The primary item of interest in this structure is **`ind->req_info_ptr`**. This is a pointer to the operation specific data structure for a DownloadSegment indication (**`DOWNLOAD_REQ_INFO`**). This pointer will always be valid when **`u_download_ind`** is called.

Return Value: `ST_VOID` (ignored)

Operation-Specific Data Structure Used: **`DOWNLOAD_REQ_INFO`**

See page 2-297 for a detailed description of this structure.

mp_download_resp

Usage: This primitive response function sends a DownloadSegment positive response PDU. This function should be called after the **u_download_ind** function is called (a DownloadSegment indication was received), and after the required segment of the download has been obtained. This service is used to send another segment of a download, initiated by the local node, when the remote node, whose domain is being downloaded, requests another segment.

Function Prototype:

```
ST_RET mp_download_resp (MMSREQ_IND *ind,
                         DOWNLOAD_RESP_INFO *info);
```

Parameters:

`ind` This pointer to the indication control data structure of type **MMSREQ_IND** is passed to the **u_download_ind** function when it was called.

`info` This pointer to an Operation Specific data structure of type **DOWNLOAD_RESP_INFO** contains information specific to the response PDU to be sent.

Return Value: `ST_RET` **SD_SUCCESS**. No Error.

`<> 0` Error Code.

Corresponding User Indication Function: **u_download_ind**

Operation-Specific Data Structure Used: **DOWNLOAD_RESP_INFO**

See page 2-297 for a detailed description of this structure.

u_mp_download_conf

Usage: This primitive user confirmation function is called when a confirm to a DownloadSegment request (**mp_download**) is received. **resp_err** contains a value indicating whether an error occurred. **resp_info_ptr** may contain data that the remote node sent back in the confirmation or it may contain error information if an error occurred. If no error has occurred (**req->resp_err = CNF_RESP_OK**), DownloadSegment information is available to the application using the **req->resp_info_ptr** pointer. See **Volume 1 — Module 1 — Primitive User Confirmation Function Class** for more information on the recommended handling of confirms.

Note that the function called for this event may be changed if desired; however, **u_mp_download_conf** is the standard default function.

Function Prototype:

```
ST_VOID u_mp_download_conf (MMSREQ_PEND *req);
```

Parameters:

`req` This pointer to the **MMSREQ_PEND** structure is returned from the original **mp_download** request. See **Volume 1 — Module 2 — Request and Indication Control Data Structures** for more information on this structure.

For a positive response (**req->resp_err = CNF_RESP_OK**), the request control structure member **resp_info_ptr** is a pointer to the operation specific data structure (**DOWNLOAD_RESP_INFO**) for the DownloadSegment function.

For a negative response, or other errors, see **Volume 3 — Module 11 — MMS-EASE Error Handling** for more information.

Return Value: `ST_VOID` (ignored)

Operation-Specific Data Structure Used: **DOWNLOAD_RESP_INFO**

See page 2-297 for a detailed description of this structure.

6. TerminateDownloadSequence Service

This service is requested by the Server to notify the Client that the download sequence is complete.

Primitive Level TerminateDownloadSequence Operations

The following section contains information on how to use the paired primitive interface for the Terminate-DownloadSequence service. It covers available data structures used by the PPI, and the four primitive level functions that together make up the TerminateDownloadSequence service. See **Volume 1 — Module 2 — General Sequence of Events — Confirmed Services** for a detailed explanation of the sequence of events that occur within the MMS-EASE environment during confirmed services.

- The TerminateDownloadSequence service consists of the paired primitive functions of **`mp_termdown`**, **`u_termdown_ind`**, **`mp_termdown_resp`**, and **`u_mp_termdown_conf`**.

Data Structures

Request/Indication

The operation specific structure described below is used by the Client in issuing a TerminateDownloadSequence request (**`mp_termdown`**). It is received by the Server when a TerminateDownloadSequence indication (**`u_termdown_ind`**) is received.

```
struct termdown_req_info
  {
  ST_CHAR    dname [MAX_IDENT_LEN +1];
  ST_BOOLEAN discarded;
  ERR_INFO   *err;
  };
typedef struct termdown_req_info TERMDOWN_REQ_INFO;
```

Fields:

`dname` This contains the name of the domain that was downloaded.

`discarded` **`SD_FALSE`**. The download was successful, and is now READY. This is the default.

`SD_TRUE`. The download was not successful, and any received download segments have been discarded. The domain has been deleted.

`err` This pointer of structure type **`ERR_INFO`** contains the service error. This is used for IS only and only if **`discarded != SD_FALSE`**. See **Volume 3 — Module 11 — Error Handling** for more information on this structure.

Paired Primitive Interface Functions

mp_termdown

Usage: This primitive request function sends a TerminateDownloadSequence request PDU using the data from a structure of type **TERMDOWN_REQ_INFO**, pointed to by **info**. This service is used to indicate that a download sequence was completed (or aborted). It was initiated via an InitiateDownloadSequence service request. This request is sent from the node that was downloaded to, not the node that initiated the download.

Function Prototype:

```
MMSREQ_PEND *mp_termdown (ST_INT chan,
                          TERMDOWN_REQ_INFO *info);
```

Parameters:

`chan` This integer contains the channel number over which the TerminateDownloadSequence PDU is to be sent.

`info` This pointer to an Operation-Specific data structure of type **TERMDOWN_REQ_INFO** contains information specific to the request PDU to be sent.

Return Value:

`MMSREQ_PEND *` This pointer to the request control data structure of type **MMSREQ_PEND** is used to send the PDU. In case of an error, the pointer is set to null and **mms_op_err** is written with the error code.

Corresponding User Confirmation Function: **u_mp_termdown_conf**

Operation-Specific Data Structure Used: **TERMDOWN_REQ_INFO**

See page 2-303 for a detailed description of this structure.

u_termdown_ind

Usage: This user indication function is called when a TerminateDownloadSequence indication is received by a Server node. The contents and operation of this function are user-defined and depend on the application requirements. To conform to the MMS protocol requirements, the application must do one of the following some time after the indication is received:

1) send a positive response using the primitive response function (**`mp_termdown_resp`**) after the download to the specified domain has been successfully terminated, or

2) call the appropriate primitive negative (error) response function (**`mp_err_resp`**). See **Volume 3 — Module 11 — MMS-EASE Error Handling** for more information on this function.

See **Volume 1 — Module 1 — User Indication Function Class** for additional information on the recommended handling of indications.

Function Prototype: `ST_VOID u_termdown_ind (MMSREQ_IND *ind);`

Parameters:

`ind` This pointer to the **`MMSREQ_IND`** structure is described in **Volume 1 — Module 2 — Request and Indication Control Data Structures**. This pointer must be used to send the response to this indication. The primary item of interest in this structure is **`ind->req_info_ptr`**. This is a pointer to the operation-specific data structure for a TerminateDownloadSequence (**`TERMDOWN_REQ_INFO`**). This pointer will always be valid when **`u_termdown_ind`** is called.

Return Value: `ST_VOID` (ignored)

Operation-Specific Data Structure Used: **`TERMDOWN_REQ_INFO`**

See page 2-303 for a detailed description of this structure.

mp_termdown_resp

Usage: This primitive response function sends a TerminateDownloadSequence positive response PDU. This function should be called as a response to the **u_termdown_ind** function being called (a TerminateDownloadSequence indication was received), and after the download to the specified domain has been successfully terminated (the domain is either discarded or ready to run) as specified by the indication.

Function Prototype: `ST_RET mp_termdown_resp (MMSREQ_IND *ind);`

Parameters:

`ind` This pointer to the indication control structure of type **MMSREQ_IND** is passed to the **u_termdown_ind** function when it was called.

Return Value: `ST_RET` **SD_SUCCESS**. No Error.

`<> 0` Error Code.

Corresponding User Indication Function: **u_termdown_ind**

Operation-Specific Data Structure Used: NONE

u_mp_termdown_conf

Usage: This primitive user confirmation function is called when a confirm to a TerminateDownloadSequence request (**mp_termdown**) is received. **resp_err** contains a value indicating whether an error occurred. **resp_info_ptr** may contain data that the remote node sent back in the confirmation or it may contain error information if an error occurred. If no error has occurred (**req->resp_err = CNF_RESP_OK**), TerminateDownloadSequence information is available to the application using the **req->resp_info_ptr** pointer. See **Volume 1 — Module 1 — Primitive User Confirmation Function Class** for more information on the recommended handling of confirms.

Note that the function called for this event may be changed if desired; however, **u_mp_termdown_conf** is the standard default function.

Function Prototype:

```
ST_VOID u_mp_termdown_conf (MMSREQ_PEND *req);
```

Parameters:

`req` This pointer to the **MMSREQ_PEND** structure is returned from the original **mp_termdown** request function. See **Volume 1 — Module 2 — Request and Indication Control Data Structures** for more information on this structure.

For a positive response (**req->resp_err = CNF_RESP_OK**), the request control structure member **resp_info_pres** indicates that there is no confirm data for this function.

For a negative response, or other errors, see **Volume 3 — Module 11 — MMS-EASE Error Handling** for more information.

Return Value: `ST_VOID` (ignored)

Operation-Specific Data Structure Used: NONE

7. Download Requestor Service

Download Requestor is not a true MMS service. Rather, it is a MMS-EASE Virtual Machine service that makes requests for domain downloads simpler for the user. It is a composite service that starts by sending a request for the InitiateDownloadSequence service to the remote node. Then, it automatically responds to subsequence incoming DownloadSegment indications from the remote node. Once all the data associated with this domain has been transferred, the remote node should send a TerminateDownloadSequence request. The MMS-EASE virtual machine will automatically send a response to this indication. The virtual machine will initialize and maintain state machines associated with the download operation.

Primitive Level Domain Download Operations

There is no paired primitive interface functionality for the Download Requestor service. However, all the MMS operations, performed by the virtual machine function for this service, can be done at the paired primitive level by calling the individual paired primitive functions to:

1. request the InitiateDownloadSequence service (**`mp_initdown`**),
2. respond to DownloadSegment indications (**`mp_download_resp`**), and
3. respond to TerminateDownloadSequence indications (**`mp_termdown_resp`**).

Virtual Machine Domain Download Operations

The following section contains information on how to use the virtual machine interface for the Download Requestor service. It covers data structures used by the VMI, and the functions that comprised this service.

- The Download Requestor service consists of the virtual machine functions of **`mv_download`** and **`u_mv_download_conf`**.

Data Structure

When the virtual machine is used to request a domain download, it stores information regarding the download in a **`MV_DL_REQ_INFO`** structure. Like operation-specific data structures, it is pointed to by the **`req_info_ptr`** of the **`MMSREQ_PEND`** structure.

```
struct mv_dl_req_info
  {
  ST_LONG asn1_byte_count;
  ST_INT  type;
  ST_INT  dl_block_size;
  union
    {
    struct
      {
      ST_LONG  bytes_left;
      ST_UCHAR *buffer;
      } bufinfo;
    struct
      {
      FILE     *fp;
      ST_UCHAR *filebuf;
      } fileinfo;
    ST_CHAR *(*dom_data_fun) (ST_INT max_len, ST_INT *ret_len_ptr,
                              ST_CHAR *more_follows_ptr);
    } i;
  };
typedef struct mv_dl_req_info MV_DL_REQ_INFO;
```

Fields:

`byte_count` This contains a running count of the bytes downloaded to the Server.

`type` This is a union tag field for union **`i`**. This value matches the value of **`src_type`**, which is a function parameter for **`mv_download`**. Possible values are:

`MEMDOM`. This indicates that the data to be downloaded is memory resident, and the **`bufinfo`** structure member of union **`i`** will be used.

`FILDOM`. This indicates that the data to be downloaded is stored in a file, and the **`fileinfo`** structure member of union **`i`** will be used.

`FUNDOM`. This indicates that the data to be downloaded will come from a user-supplied function, and the **`dom_data_fun`** function pointer member of union **`i`** will be used.

`dl_block_size` This contains the maximum size of each data element sent in a Download Segment response PDU.

`bytes_left` This indicates the number of remaining bytes to be downloaded. Used if **`type = MEMDOM`**.

`buffer` This pointer to the download data buffer is used if **`type = MEMDOM`**.

`fp` This pointer to the operating system file stream control structure provides access to the file containing the downloaded data. Used if **`type = FILDOM`**. The file must already be opened for read operations (binary mode is recommended).

`filebuf` This pointer to the Virtual Machine data buffer is used if **`type = FILDOM`**.

`dom_data_fun` This pointer to the user-defined function provides the download data. Used if **`type = FUNDOM`**.

Virtual Machine Interface Functions

mv_download

Usage: This virtual machine function allows the user to execute the entire download sequence automatically. This includes an InitiateDownloadSequence, multiple responses to DownloadSegment indications, and a response to a TerminateDownloadSequence indication. It is accomplished without requiring the user to generate and manage the individual requests, confirmations, responses, etc., necessary to download domains. Also, the necessary state information regarding this domain download is initialized and maintained by the virtual machine automatically in **MV_DL_REQ_INFO**. See page 2-309 for a detailed description of this structure.

Functional Prototype:

```
MMSREQ_PEND *mv_download (ST_INT chan,
                          ST_CHAR *dom_name,
                          ST_INT num_cap,
                          ST_CHAR **cap,
                          ST_INT src_type,
                          ST_LONG src_len,
                          ST_UCHAR *src);
```

Parameters:

`chan` This is the channel number over which the request PDU is to be sent.

`dom_name` This is a pointer to the domain name being downloaded.

`num_cap` This indicates the number of capabilities (pointers) in the **cap** array.

`cap` This is the address of an array of pointers to implementation-specific null-terminated character strings describing the limitations on, and capabilities of, the downloaded domain.

`src_type` This specifies the type of domain being downloaded:

MEMDOM. The downloaded domain data is memory resident.
FILDOM. The downloaded domain data is contained in a file.
FUNDOM. The downloaded domain data will come from a user-supplied function.

`src_len` This indicates, in bytes, the length of the domain being download. Used only if **src_type = MEMDOM**.

`src` This is a pointer to the domain data to be downloaded. Its type will actually depend on the value in **src_type**. If **src_type = MEMDOM**, **src** will be a ****ST_CHAR** pointer to the memory resident data. If **src_type = FILDOM**, **src** will be a **FILE** pointer to a user-defined function that will provide the domain data. See Note #3 on page 2-313 for more information.

Return Value:

`MMSREQ_PEND *` This pointer to the request control data structure of type **MMSREQ_PEND** is used to send the PDU. Data pertaining to this request is stored in the structure **MV_DL_REQ_INFO**. In case of an error, the pointer is returned **= 0** and **mms_op_err** is written with the error code.

Corresponding User Confirmation Function: **u_mv_download_conf**

u_mv_download_conf

Usage: This virtual machine user confirmation function is called when a confirm to a virtual machine request (**mv_download**) is received. **resp_err** contains a value indicating whether an error occurred. **resp_info_ptr** may contain data that the remote node sent back in the confirmation or it may contain error information if an error occurred. If no error has occurred (**req->resp_err = CNF_RESP_OK**), Download information is available to the application using the **req->resp_info_ptr** pointer. See **Volume 1 — Module 1 — Virtual User Confirmation Function Class** for more information on the recommended handling of confirms.

Note that the function called for this event may be changed if desired; however, **u_mv_download_conf** is the standard default function.

Function Prototype:

```
ST_VOID u_mv_download_conf (MMSREQ_PEND *req);
```

Parameters:

`req` This pointer to the request control data structure of type **MMSREQ_PEND** is used to send the initial request (returned from **mv_download**). See **Volume 1 — Module 2 — Request and Indication Control Data Structures** for more information on this structure.

Return Value: `ST_VOID` (ignored)

NOTES:

1. Before returning from any virtual user confirmation function, **ms_clr_mvreq** should be called. It will clear up and free the data used by the virtual machine to handle the request and confirmation.
2. The contents of this function are completely user-defined.

Special Application Notes:

1. The application can monitor the progress of the domain download by examining the **byte_count** member of the **MV_DL_REQ_INFO** structure.

   WARNING!!! The pointer to **MV_DL_REQ_INFO** is only valid until the **u_mv_download_conf** function is returned from. After this, the pointer may be returned to a free pool during a call to **ms_comm_serve**.

2. The virtual machine automatically generates response PDUs to the Download Segment requests until it receives as a result of requesting a domain download to a remote node. The size of the data segment to be downloaded in each response is determined by the value in **mms_chan_info[chan].download_blk_size**. The default value for this size is negotiated at the time the association is established for this channel. However, the user may change this value before calling **mv_download**, if a difference size is desired.

3. If **mv_download** gets the data from a user-supplied function (i.e., **src_type = FUNDOM**), that function will be called as follows:

```
ST_CHAR *user_fun (max_len,ret_len_ptr,more_follows_ptr)
ST_INT  max_len;
ST_INT  *ret_len_ptr;
ST_CHAR *more_follows_ptr;
   {
    user defined function
   };
```

   Fields:

   max_len — This indicates the maximum number of bytes that the virtual machine will accept from the user function.

   ret_len_ptr — This is a pointer to the number of bytes that are physically returned from the function. This number should be <= to the value in **max_len**.

   more_follows_ptr — This is a pointer to a **more_follows** flag.

   If this flag is set to **SD_TRUE**, MMS-EASE will call the user function again to retrieve more download data.

   If this flag is set to **SD_FALSE**, MMS-EASE will not call the user function again to retrieve more download data.

   The return value is a pointer to the downloaded data. A null pointer implies that there is no more data to be downloaded.

   When the virtual machine receives a TerminateDownloadSequence request from the remote node, this user function will be called. It will verify that there is no more data to be downloaded. If the function does not return a null pointer, the virtual machine will respond to the TerminateDownloadSequence request with a **DOM_STATE** error.

mv_forced_download

Usage: This virtual machine function allows the application to perform a download operation to a remote node from either a local file, local buffer, or local function that could result in destructive operations taking place on the server. Any existing Domain on the server will be deleted before the new Domain is downloaded. Should the Domain exist and have some Program Invocations referencing it, they are killed and deleted; then the Domain is deleted before the download takes place.

Functional Prototype:

```
MMSREQ_PEND *mv_forced_download (ST_INT chan,
                                 ST_CHAR *dom_name,
                                 ST_INT num_cap,
                                 ST_CHAR **cap,
                                 ST_INT src_type,
                                 ST_LONG src_len,
                                  ST_UCHAR *src);
```

Parameters:

`chan` This is the channel number over which the Download will take place.

`dom_name` This is a pointer to the domain name being forced to download.

`num_cap` This indicates the number of capabilities (pointers) in the **`cap`** array.

`cap` This is the address of an array of pointers to implementation-specific null-terminated character strings describing the limitations on, and capabilities of, the downloaded domain.

`src_type` This specifies the type of domain being forced to download:

`MEMDOM`. The downloaded domain data is memory resident.
`FILDOM`. The downloaded domain data is contained in a file.
`FUNDOM`. The downloaded domain data will come from a user-supplied function.

`src_len` This indicates, in bytes, the length of the domain being forced to download. Used only if **`src_type = MEMDOM`**.

`src` This is a pointer to the domain data to be downloaded. Its type will actually depend on the value in **`src_type`**. If **`src_type = MEMDOM`**, **`src`** will be a **`**ST_CHAR`** pointer to the memory resident data. If **`src_type = FILDOM`**, **`src`** will be a **`FILE`** pointer to a user-defined function that will provide the domain data. See Note #3 on page 2-313 for more information.

Return Value:

`MMSREQ_PEND *` This pointer to the request control data structure of type **`MMSREQ_PEND`** is used to send the PDU. Data pertaining to this request is stored in the structure **`MV_DL_REQ_INFO`**. In case of an error, the pointer is set to null and **`mms_op_err`** is written with the error code.

8. InitiateUploadSequence Service

This service is used by the Client to request that the Server prepare to upload the specified named Domain.

Primitive Level InitiateUploadSequence Operations

The following section contains information on how to use the paired primitive interface for the InitiateUploadSequence service. It covers available data structures used by the PPI, and the four primitive level functions that together make up the InitiateUploadSequence service. See **Volume 1 — Module 2 — General Sequence of Events — Confirmed Services** for a detailed explanation of the sequence of events that occur within the MMS-EASE environment during confirmed services.

- The InitiateUploadSequence service consists of the paired primitive functions of **`mp_initupl`**, **`u_initupl_ind`**, **`mp_initupl_resp`**, and **`u_mp_initupl_conf`**.

Data Structures

Request/Indication

The operation specific structure described below is used by the Client in issuing an InitiateUploadSequence request (**`mp_initupl`**). It is received by the Server when an InitiateUploadSequence indication (**`u_initupl_ind`**) is received.

```
struct initupl_req_info
  {
  ST_CHAR dname [MAX_IDENT_LEN +1];
  };
typedef struct initupl_req_info  INITUPL_REQ_INFO;
```

Fields:

`dname` This contains the name of the domain to be uploaded.

Response/Confirm

This operation specific data structure described below is used by the Server in issuing an InitiateUploadSequence response (**mp_initupl_resp**). It is received by the Client when an InitiateUploadSequence confirm (**u_mp_initupl_conf**) is received.

```
struct initupl_resp_info
  {
  ST_INT32 ulsmid;
  ST_INT   num_of_capab;
/*ST_CHAR  *capab_list [num_of_capab];                    */
  SD_END_STRUCT
  };
typedef struct initupl_resp_info INITUPL_RESP_INFO;
```

Fields:

`ulsmid` This contains the Upload State Machine ID. This number is used to identify this particular upload from other uploads during subsequent UploadSegment services.

`num_capab` This indicates the number of elements in the **capab_list** array.

`capab_list` This array of pointers to implementation specific null-terminated character strings describes the limitations on, and capabilities of, the VMD that are a part of the domain to be uploaded.

NOTE: FOR RESPONSE ONLY, when allocating a data structure of type **INITUPL_RESP_INFO**, enough memory must be allocated to hold the information for the capabilities list contained in **capab_list**. The following C statement can be used:

```
info = (INITUPL_RESP_INFO *) chk_malloc(sizeof (INITUPL_RESP_INFO) + (num_capab *
             (sizeof(ST_CHAR *))));
```

Paired Primitive Interface Functions

mp_initupl

Usage: This primitive request function sends an InitiateUploadSequence request PDU using the data from a structure of type **INITUPL_REQ_INFO**, pointed to by **info**. The InitiateUploadSequence service is used to begin uploading a domain from a remote node.

Function Prototype:
```
MMSREQ_PEND *mp_initupl (ST_INT chan,                INITUPL_REQ_INFO *info);
```

Parameters:

chan	This integer contains the channel number over which the InitiateUploadSequence PDU is to be sent.
info	This pointer to an Operation-Specific data structure of type **INITUPL_REQ_INFO** contains information specific to the request PDU to be sent.

Return Value:

MMSREQ_PEND *	This pointer to the request control data structure of type **MMSREQ_PEND** is used to send the PDU. In case of an error, the pointer is set to null and **mms_op_err** is written with the error code.

Corresponding User Confirmation Function: **u_mp_initupl_conf**

Operation-Specific Data Structure Used: **INITUPL_REQ_INFO**

See page 2-315 for a detailed description of this structure.

u_initupl_ind

Usage: This user indication function is called when an InitiateUploadSequence indication is received by a Server node. The contents and operation of this function are user-defined and depend on the application requirements. To conform to the MMS protocol requirements, the application must do one of the following some time after the indication is received:

1) send a positive response using the primitive response function (**`mp_initupl_resp`**) after successfully preparing for the upload, or

2) call the appropriate primitive negative (error) response function (**`mp_err_resp`**). See **Volume 3 — Module 11 — MMS-EASE Error Handling** for more information on this function.

See **Volume 1 — Module 1— User Indication Function Class** for additional information on the recommended handling of indications.

Function Prototype:

```
ST_VOID u_initupl_ind (MMSREQ_IND *ind);
```

Parameters:

`ind` This pointer to the **`MMSREQ_IND`** structure is described in **Volume 1 — Module 2 — Request and Indication Control Data Structures**. This pointer must be used to send the response to this indication. The primary item of interest in this structure is **`ind->req_info_ptr`**. This is a pointer to the operation-specific data structure for an InitiateUploadSequence indication (**`INITUPL_REQ_INFO`**). This pointer is always valid when an **`u_initupl_ind`** is called.

Return Value: `ST_VOID` (ignored)

Operation-Specific Data Structure Used: **`INITUPL_REQ_INFO`**

See page 2-315 for a detailed description of this structure.

mp_initupl_resp

Usage: This primitive response function sends an InitiateUploadSequence positive response PDU using the data from a structure of type **INITUPL_RESP_INFO**, pointed to by **info**. This function should be called as a response to the **u_initupl_ind** function being called (an InitiateUploadSequence indication was received), and after successfully preparing for the upload.

Function Prototype:

```
ST_RET mp_initupl_resp (MMSREQ_IND *ind,
                        INITUPL_RESP_INFO *info);
```

Parameters:

`ind` This pointer to the indication control data structure of type **MMSREQ_IND** is passed to the **u_initupl_ind** function when it was called.

`info` This pointer to an Operation-Specific data structure of type **INITUPL_RESP_INFO** contains information specific to the response PDU to be sent.

Return Value: `ST_RET`

SD_SUCCESS. No Error.

`<> 0` Error Code.

Corresponding User Indication Function: **u_initupl_ind**

Operation-Specific Data Structure Used: **INITUPL_RESP_INFO**

See page 2-315 for a detailed description of this structure.

u_mp_initupl_conf

Usage: This primitive user confirmation function is called when a confirm to an InitiateUploadSequence request (**mp_initupl**) is received. **resp_err** contains a value indicating whether an error occurred. **resp_info_ptr** may contain data that the remote node sent back in the confirmation or it may contain error information if an error occurred. If no error has occurred (**req->resp_err = CNF_RESP_OK**), InitiateUploadSequence information is available to the application using the **req->resp_info_ptr** pointer. See **Volume 1 — Module 1 — Primitive User Confirmation Function Class** for more information on the recommended handling of confirms.

Note that the function called for this event may be changed if desired; however, **u_mp_initupl_conf** is the standard default function.

Function Prototype:

```
ST_VOID u_mp_initupl_conf (MMSREQ_PEND *req);
```

Parameters:

`req` This pointer to the **MMSREQ_PEND** structure is returned from the original **mp_initupl** request function. See **Volume 1 — Module 2 — Request and Indication Control Data Structures** for more information on this structure.

For a positive response (**req->resp_err = CNF_RESP_OK**), the request control structure member **resp_info_ptr** is a pointer to the operation specific data structure (**INITUPL_RESP_INFO**) for the InitiateUploadSequence function.

For a negative response, or other errors, see **Volume 3 — Module 11 — MMS-EASE Error Handling** for more information.

Return Value: `ST_VOID` (ignored)

Operation-Specific Data Structure Used: **INITUPL_RESP_INFO**

See page 2-315 for a detailed description of this structure.

9. UploadSegment Service

This service is requested by the Client to have a segment of upload data transferred from the Server.

Primitive Level UploadSegment Operations

The following section contains information on how to use the paired primitive interface for the UploadSegment service. It covers available data structures used by the PPI, and the four primitive level functions that together make up the UploadSegment service. See **Volume 1 — Module 2 — General Sequence of Events - Confirmed Services** for a detailed explanation of the sequence of events that occur within the MMS-EASE environment during confirmed services.

- The UploadSegment service consists of the paired primitive functions of **`mp_upload`**, **`u_upload_ind`**, **`mp_upload_resp`**, and **`u_mp_upload_conf`**.

Data Structures

Request/Indication

The operation specific structure described below is used by the Client in issuing an UploadSegment request (**`mp_upload`**). It is received by the Server when an UploadSegment indication (**`u_upload_ind`**) is received.

```
struct upload_req_info
  {
  ST_INT32   ulsmid;
  };
typedef struct upload_req_info UPLOAD_REQ_INFO;
```

Fields:

`ulsmid` This contains the **U**p**l**oad **S**tate **M**achine **ID** (ULSMID) for the domain being uploaded. The ULSMID is obtained from the InitiateUploadSequence response.

Response/Confirm

This operation specific data structure described below is used by the Server in issuing an UploadSegment response (**mp_upload_resp**). It is received by the Client when an UploadSegment confirm (**u_mp_upload_conf**) is received.

```
struct upload_resp_info
  {
  ST_INT     load_data_type;
  ST_INT     load_data_len;
  ST_UCHAR    *load_data;
  ST_BOOLEAN more_follows;
  };
typedef struct upload_resp_info  UPLOAD_RESP_INFO;
```

Fields:

load_data_type	This indicates the format for the data to be downloaded: **LOAD_DATA_NON_CODED**. This is an octet string. **LOAD_DATA_CODED**. This is a value encoded by some other external source ASN.1 EXTERNAL data element. See page 2-201 for more information on how to encode and decode ASN.1 EXTERNAL data elements.
load_data_len	This is the length, in bytes, of the data pointed to by **load_data**.
load_data	This is a pointer to the data segment to be uploaded.
more_follows	**SD_FALSE**. There is NO MORE DATA to be transmitted. **SD_TRUE**. There is more data to be transmitted in order to complete the upload. This is the default.

Paired Primitive Interface Functions

mp_upload

Usage: This primitive request function sends an UploadSegment request PDU using the data from a structure of type **UPLOAD_REQ_INFO**, pointed to by **info**. The UploadSegment service is used to request that the remote node transfer another segment of the domain being uploaded from the remote node.

Function Prototype:

```
MMSREQ_PEND *mp_upload (ST_INT chan,
                        UPLOAD_REQ_INFO *info);
```

Parameters:

chan	This is the channel number over which the Upload Segment request is to be sent.
info	This pointer to an Operation-Specific data structure of type **UPLOAD_REQ_INFO** contains data specific to the PDU to be sent.

Return Value:

MMSREQ_PEND *	This pointer to the request control data structure of type **MMSREQ_PEND** is used to send the PDU. In case of an error, the pointer is set to null and **mms_op_err** is written with the error code.

Corresponding User Confirmation Function: **u_mp_upload_conf**

Operation-Specific Data Structure Used: **UPLOAD_REQ_INFO**

See page 2-321 for a detailed description of this structure.

u_upload_ind

Usage: This user indication function is called when an UploadSegment indication is received by a Server node. The contents and operation of this function are user-defined and depend on the application requirements. To conform to the MMS protocol requirements, the application must do one of the following some time after the indication is received:

1) send a positive response using the primitive response function (**mp_upload_resp**) after the next segment of the specified domain (identified by the Upload State Machine ID (ulsmid)) has been successfully obtained, or

2) call the appropriate primitive negative (error) response function (**mp_err_resp**). See **Volume 3 — Module 11 — MMS-EASE Error Handling** for more information on this function.

See **Volume 1 — Module 1 — User Indication Function Class** for additional information on the recommended handling of indications.

Function Prototype:

```
ST_VOID u_upload_ind (MMSREQ_IND *ind);
```

Parameters:

`ind` This pointer to the **MMSREQ_IND** structure is described in **Volume 1 — Module 2 — Request and Indication Control Data Structures**. This pointer must be used to send the response to this indication. The primary item of interest in this structure is **ind->req_info_ptr**. This is a pointer to the operation-specific data structure for the UploadSegment indication (**UPLOAD_REQ_INFO**). This pointer will always be valid when **u_upload_ind** is called.

Return Value: `ST_VOID` (ignored)

Operation-Specific Data Structure Used: **UPLOAD_REQ_INFO**

See page 2-321 for a detailed description of this structure.

mp_upload_resp

Usage: This primitive response function sends an UploadSegment positive response PDU. This function should be called after the **u_upload_ind** function is called (an UploadSegment indication was received), and after the next segment of the specified domain has been successfully obtained. This is identified by the Upload State Machine ID (ulsmid).

Function Prototype:

```
ST_RET mp_upload_resp (MMSREQ_IND *ind,
                       UPLOAD_RESP_INFO *info);
```

Parameters:

`ind` This pointer to the indication control structure of type **MMSREQ_IND** is passed to the **u_upload_ind** function when it was called.

`info` This pointer to an Operation-Specific data structure of type **UPLOAD_RESP_INFO** contains information specific to the response PDU to be sent.

Return Value:

`ST_RET` **SD_SUCCESS**. No Error.

`<> 0` Error Code.

Corresponding User Indication Function: **u_upload_ind**

Operation-Specific Data Structure Used: **UPLOAD_RESP_INFO**

See page 2-321 for a detailed description of this structure.

u_mp_upload_conf

Usage: This primitive user confirmation function is called when a confirm to an UploadSegment request (**mp_upload**) is received. **resp_err** contains a value indicating whether an error occurred. **resp_info_ptr** may contain data that the remote node sent back in the confirmation or it may contain error information if an error occurred. If no error has occurred (**req->resp_err = CNF_RESP_OK**), UploadSegment information is available to the application using the **req->resp_info_ptr** pointer. See **Volume 1 — Module 1 — Primitive User Confirmation Function Class** for more information on the recommended handling of confirms.

Note that the function called for this event may be changed if desired; however, **u_mp_upload_conf** is the standard default function.

Function Prototype:

```
ST_VOID u_mp_upload_conf (MMSREQ_PEND *req);
```

Parameters:

`req` This pointer to the **MMSREQ_PEND** structure is returned from the original **mp_upload** request function. See **Volume 1 — Module 2 — Request and Indication Control Data Structures** for more information on this structure.

For a positive response (**req->resp_err = CNF_RESP_OK**), the request control structure member **resp_info_ptr** is a pointer to the operation-specific data structure (**UPLOAD_RESP_INFO**) for the UploadSegment function.

For a negative response, or other errors, see **Volume 3 — Module 11 — MMS-EASE Error Handling** for more information.

Return Value: `ST_VOID` (ignored)

Operation-Specific Data Structure Used: **UPLOAD_RESP_INFO**

See page 2-321 for a detailed description of this structure.

10. TerminateUploadSequence Service

This service is used by the Client to request that the Server terminate an upload sequence. This causes the Upload State Machine to be deleted, regardless of whether this service was completed successfully.

Primitive Level TerminateUploadSequence Operations

The following section contains information on how to use the paired primitive interface for the TerminateUploadSequence service. It covers available data structures used by the PPI, and the four primitive level functions that together make up the TerminateUploadSequence service. See **Volume 1 — Module 2 — General Sequence of Events — Confirmed Services** for a detailed explanation of the sequence of events that occur within the MMS-EASE environment during confirmed services.

- The TerminateUploadSequence service consists of the paired primitive functions of **`mp_termupl`**, **`u_termupl_ind`**, **`mp_termupl_resp`**, and **`u_mp_termupl_conf`**.

Data Structures

Request/Indication

The operation specific structure described below is used by the Client in issuing a TerminateUploadSequence request (**`mp_termupl`**). It is received by the Server when a TerminateUploadSequence indication (**`u_termupl_ind`**) is received.

```
struct termupl_req_info
  {
  ST_INT32   ulsmid;
  };
typedef struct termupl_req_info  TERMUPL_REQ_INFO;
```

Fields:

`ulsmid` This contains the Upload State Machine ID (ULSMID), for the domain upload sequence that has completed or aborted. The ULSMID is obtained from the InitiateUploadSequence response.

Paired Primitive Interface Functions

mp_termupl

Usage: This primitive request function sends a TerminateUploadSequence request PDU using the data from a structure of type **TERMUPL_REQ_INFO**, pointed to by **info**. This service is used to terminate an upload sequence completed, and previously initiated via an InitiateUploadSequence request. Use this service to terminate either a successful or unsuccessful (aborted) upload sequence.

Function Prototype:
```
MMSREQ_PEND *mp_termupl (ST_INT chan,
                         TERMUPL_REQ_INFO *info);
```

Parameters:

`chan` This integer contains the channel number over which the TerminateUploadSequence PDU is to be sent.

`info` This pointer to an Operation-Specific data structure of type **TERMUPL_REQ_INFO** contains information specific to the request PDU to be sent.

Return Value:

`MMSREQ_PEND *` This pointer to the request control data structure of type **MMSREQ_PEND** is used to send the PDU. In case of an error, the pointer is set to null and **mms_op_err** is written with the error code.

Corresponding User Confirmation Function: **u_mp_termupl_conf**

Operation-Specific Data Structure Used: **TERMUPL_REQ_INFO**

See page 2-327 for a detailed description of this structure.

u_termupl_ind

Usage: This user indication function is called when a TerminateUploadSequence indication is received by a Server node. The contents and operation of this function are user-defined and depend on the application requirements. To conform to the MMS protocol requirements, the application must do one of the following some time after the indication is received:

1) send a positive response using the primitive response function (**`mp_termupl_resp`**) after the upload sequence has been successfully terminated, or

2) call the appropriate primitive negative (error) response function (**`mp_err_resp`**). See **Volume 3 — Module 11 — MMS-EASE Error Handling** for more information on this function.

See **Volume 1 — Module 1 — User Indication Function Class** for additional information on the recommended handling of indications.

Function Prototype: `ST_VOID u_termupl_ind (MMSREQ_IND *ind);`

Parameters:

`ind` This pointer to the **`MMSREQ_IND`** structure is described in **Volume 1 — Module 2 — Request and Indication Control Data Structures**. This pointer must be used to send the response to this indication. The primary item of interest in this structure is **`ind->req_info_ptr`**. This is a pointer to the operation-specific data structure for a TerminateUploadSequence indication (**`TERMUPL_REQ_INFO`**). This pointer is always valid when **`u_termupl_ind`** is called.

Return Value: `ST_VOID` (ignored)

Operation-Specific Data Structure Used: **`TERMUPL_REQ_INFO`**

See page 2-327 for a detailed description of this structure.

mp_termupl_resp

Usage: This primitive response function sends a TerminateUploadSequence positive response PDU. This function should be called as a response to the **u_termupl_ind** function being called (a TerminateUploadSequence indication was received), and after the upload sequence has been successfully terminated, and the Upload State Machine ID (ULSMID) is freed up for use by other upload service requests. A result (+) response should be sent as long as the upload was terminated; regardless of whether the entire upload sequence was completed successfully.

Function Prototype:

```
ST_RET mp_termupl_resp (MMSREQ_IND *ind);
```

Parameters:

ind	This pointer to the indication control structure of type **MMSREQ_IND** is passed to the **u_termupl_ind** function when it was called.

Return Value:

ST_RET	**SD_SUCCESS**.	No Error.
	<> 0	Error Code.

Corresponding User Indication Function: **u_termupl_ind**

Operation-Specific Data Structure Used: NONE

u_mp_termupl_conf

Usage: This primitive user confirmation function is called when a confirm to a TerminateUploadSequence request (**mp_termupl**) is received. **resp_err** contains a value indicating whether an error occurred. **resp_info_ptr** may contain data that the remote node sent back in the confirmation or it may contain error information if an error occurred. See **Volume 1— Module 1 — Primitive User Confirmation Function Class** for more information on the recommended handling of confirms.

Note that the function called for this event may be changed if desired; however, **u_mp_termupl_ind** is the standard default function.

Function Prototype:

```
ST_VOID u_mp_termupl_conf (MMSREQ_PEND *req);
```

Parameters:

`req` This pointer to the **MMSREQ_PEND** structure is returned from the original **mp_termupl** request function. See **Volume 1 — Module 2 — Request and Indication Control Data Structures** for more information on this structure.

For a positive response (**req->resp_err = CNF_RESP_OK**), the request control structure member **resp_info_pres** indicates that there is no confirm data for this function.

For a negative response, or other errors, see **Volume 3 — Module 11 — MMS-EASE Error Handling** for more information.

Return Value: `ST_VOID` (ignored)

Operation-Specific Data Structure Used: NONE

11. Upload Requestor Service

Upload Requestor is not a true MMS service. Rather, it is a MMS-EASE Virtual Machine service that makes requests for domain uploads simpler for the user. It is a composite service that starts by sending a request for the InitiateUploadSequence service to the remote node. Then, it automatically responds to subsequent incoming UploadSegment indications from the remote node. Once all the data associated with this domain has been transferred, the remote node should send a TerminateUploadSequence request. The virtual machine will initialize and maintain state machines associated with the upload operation.

Primitive Level Domain Upload Operations

There is no paired primitive interface functionality for the Upload Requestor service. However, all the MMS operations performed by the virtual machine function for this service can be done at the paired primitive level by calling the individual paired primitive functions to:

1. request the InitiateUploadSequence service (**`mp_initupl`**),
2. respond to UploadSegment indications (**`mp_upload_resp`**), and
3. respond to TerminateUploadSequence indications (**`mp_termupl_resp`**).

Virtual Machine Domain Upload Operations

The following section contains information on how to use the virtual machine interface for the Upload Requestor service. It covers the data structures and functions used by the VMI.

- The Upload Requestor service consists of the virtual machine functions of **`mv_upload`** and **`u_mv_upload_conf`**.

Data Structures

When the virtual machine is used to request a domain upload, it stores information regarding the upload in the following data structure. Like the Operation-Specific data structures, it is pointed to by the **`req_info_ptr`** of the **`MMSREQ_PEND`** structure.

```
struct mv_ul_req_info
  {
  ST_LONG  asn1_byte_count;
  ST_INT   type;
  ST_INT32 ulsmid;
  ST_INT   num_of_capab;
  union
    {
    struct
      {
      ST_LONG bytes_left;
      ST_UCHAR *buffer;
      } bufinfo;
    struct
      {
      FILE *fp;
      } fileinfo;
    ST_RET (*dom_data_fun)(struct upload_resp_info *rsp_info);
    } i;
/*ST_CHAR *capab_list [num_of_capab];                    */
/*SD_END_STRUCT                                              */
  };
```

```
typedef struct mv_ul_req_info MV_UL_REQ_INFO;
```

Fields:

`byte_count` This contains a running count of the bytes uploaded to the Server.

`type` This is a union tag field for union **`i`**. The value matches the value of **`src_type`**. It is also a function parameter for **`mv_upload`**. Possible values are:

`MEMDOM` indicates that the data to be uploaded is memory resident, and the **`bufinfo`** structure member of union **`i`** will be used.

`FILDOM` indicates that the data to be uploaded is stored in a file, and the **`fileinfo`** structure member of union **`i`** will be used.

`FUNDOM` indicates that the data to be uploaded will come from a user-supplied function, and the **`dom_data_fun`** function pointer member of union **`i`** will be used.

`ulsmid` This is the Upload State Machine ID for the upload sequence. It is valid only after receipt of a positive initiate upload confirm PDU.

`capab_list` This array contains pointers to implementation-specific null-terminated character strings. They describe limitations on the resources and capabilities of the VMD that are to be a part of the downloaded domain.

`bytes_left` This indicates the number of remaining bytes available in the user-supplied buffer. Used only if **`type = MEMDOM`**.

`buffer` This is the pointer to the data buffer where the next upload data segment will be written. Used only if **`type = MEMDOM`**.

`fp` This pointer to the operating system file stream control structure provides access to the file that is receiving the uploaded data. Used only if **`type = FILDOM`**. The file must already be opened for write operations (binary mode is recommended).

`dom_data_fun` This is a pointer to the user-defined function used to receive the upload data. Used only if **`type = FUNDOM`**.

`capab_list` This array of pointers to implementation-specific null-terminated character strings describes limitations on the resources and capabilities of the VMD that are to be a part of the downloaded domain.

NOTE: When allocating a data structure of type **`MV_UL_REQ_INFO`**, enough memory must be allocated to hold the pointer for the capabilities list contained in **`capab_list`**. The following C statement can be used:

```
info = (MV_UL_REQ_INFO*) chk_malloc (sizeof (MV_UL_REQ_INFO) + (num_capab *
            (sizeof(ST_CHAR *))));
```

Virtual Machine Interface Functions

mv_upload

Usage: This virtual machine function allows the user to execute the entire upload sequence automatically. This includes an InitiateUploadSequence request, multiple responses to UploadSegment indications, and a TerminateUploadSequence request. This is accomplished without requiring the user to generate and manage the individual requests, confirmations, and responses etc. necessary to upload domains. Furthermore, the necessary state information regarding this domain upload is initialized and maintained by the virtual machine automatically in **MV_UL_REQ_INFO**.

Function Prototype:

```
MMSREQ_PEND *mv_upload   (ST_INT chan,
                          ST_CHAR *dom_name,
                          ST_INT dest_type,
                          ST_LONG dest_len,
                          ST_UCHAR *dest);
```

Parameters:

`chan` This is the channel number over which the request PDU is to be sent.

`dom_name` This is a pointer to the domain name being uploaded.

`dest_type` This specifies the type of domain being uploaded, as follows:

MEMDOM. The uploaded domain data will become memory resident at the local node.\

FILDOM. The uploaded domain data will be written to a file.

FUNDOM. The uploaded domain data will be passed to a user-defined function.

`dest_len` This indicates, in bytes, the size of the domain being uploaded.

`dest` This is a pointer to where the domain data will go as it is uploaded. Its type depends on the value in **dest_type**. If **dest_type = MEMDOM**, **dest** is a pointer to the operating system file stream control structure. This provides access to the file that will receive the upload data segments. If **dest_type = FUNDOM**, **dest** is a pointer to a user-defined function that will receive the data. See note #3 on page 2-337 for more information.

Return Value:

`MMSREQ_PEND *` This pointer to the request control data structure of type **MMSREQ_PEND** is used to send the PDU. In case of an error, the pointer is set to null and **mms_op_err** is written with the error code.

Corresponding User Confirmation Function: **u_mv_upload_conf**

u_mv_upload_conf

Usage: This virtual machine user confirmation function is called when a confirm to a virtual machine request (**mv_upload**) is received. **resp_err** contains a value indicating whether an error occurred. **resp_info_ptr** may contain data that the remote node sent back in the confirmation or it may contain error information if an error occurred. See **Volume 1 — Module 1 — Virtual User Confirmation Function Class** for more information on the recommended handling of confirms.

Note that the function called for this event may be changed if desired; however, **u_mv_upload_conf** is the standard default function.

Function Prototype: `ST_VOID u_mv_upload_conf (MMSREQ_PEND *req);`

Parameters:

`req` This pointer to the **MMSREQ_PEND** structure is returned from the original **mv_upload** request. See **Volume 1 — Module 2 — Request and Indication Control Data Structures** - for more information on this structure.

For a positive response (**req->resp_err = CNF_RESP_OK**), the request control structure member **req_info_ptr** is a pointer to the operation-specific data structure (**MV_UL_REQ_INFO**). See page 2-261.

For a negative response, or other errors, see **Volume 3 — Module 11 — MMS-EASE Error Handling** - for more information.

Return Value: `ST_VOID` (ignored)

NOTES:

1. Before returning from any virtual user confirmation function, **ms_clr_mvreq** should be called. This will clear up and free the data used by the virtual machine to handle the request and confirmation.
2. The contents of this function are completely user-defined.
3. The following is an example of how to access the capabilities that are now available when the upload is completed successfully:

```
ST_VOID print_cap()
 {
 MV_UL_REQ_INFO  *ul_req_info;
 ST_CHAR         **cap_ptr;
 ST_INT          i;
 ul_req_info=(struct mv_ul_req_info *)req->req_info_ptr;
        printf ("\n Capabilities :");
        cap_ptr = (ST_CHAR **) (ul_req_info+1);
        for (i = 0;i < ul_req_info->num_of_capab; ++i, ++cap_ptr)
          printf ("\n  %s",*cap_ptr);
 };
```

Special Application Notes

1. The application can monitor the progress of the domain upload by examining the **byte_count** member of **MV_UL_REQ_INFO** structure.

   WARNING!!! The pointer to **MV_UL_REQ_INFO** is only valid until the **u_mv_upload_conf** is returned from. After this, the pointer may be returned to the free pool during a call to **ms_comm_serve**.

2. If **mv_upload** passes the data to a user-defined function (i.e., **dest_type = FUNDOM**), that function will be called as follows:

```
ST_RET user_fun (rsp_info)
UPLOAD_RESP_INFO *rsp_info;
{
user defined function
}
```

   Parameters:

   `rsp_info` This is a pointer to an upload response structure of type **UPLOAD_RESP_INFO**. See page 2-333 for a detailed description of this structure.

   The return value is a **ST_RET** that indicates the **SD_SUCCESS** or **SD_FAILURE** of the function:

   SD_SUCCESS. No Error.

   `<> 0` An error code that causes the virtual machine to terminate the upload sequence, and returns the error to the user in the confirmation function.

12. RequestDomainDownload Service

This service is used by a VMD (Server) to request that the Client perform a download of a domain in the VMD.

Primitive Level RequestDomainDownload Operations

The following section contains information on how to use the paired primitive interface for the RequestDomainDownload service. It covers available data structures used by the PPI, and the four primitive level functions that together make up the RequestDomainDownload service. See **Volume 1 — Module 2 — General Sequence of Events — Confirmed Services** for a detailed explanation of the sequence of events that occur within the MMS-EASE environment during confirmed services.

- The RequestDomainDownload service consists of the paired primitive functions of **mp_rddwn**, **u_rddwn_ind**, **mp_rddwn_resp**, and **u_mp_rddwn_conf**.

Data Structures

Request/Indication

The operation specific structure described below is used by the Client in issuing a RequestDomainDownload request (**mp_rddwn**). It is received by the Server when a RequestDomainDownload indication (**u_rddwn_ind**) is received.

```
struct rddwn_req_info
  {
  ST_CHAR    dname [MAX_IDENT_LEN +1];
  ST_BOOLEAN sharable;
  ST_INT     num_of_capab;
  ST_INT     num_of_fname;
/*ST_CHAR    *capab_list [num_of_capab];                          */
/*FILE_NAME  fname_list[ num_of_fname ];                          */
  SD_END_STRUCT
  };
typedef struct rddwn_req_info RDDWN_REQ_INFO;
```

Fields:

dname — This contains the name of the local domain to be downloaded.

sharable — **SD_FALSE**. After downloading the specified domain, it may be used by more than one Program Invocation concurrently.

SD_TRUE. After downloading the specified domain, it may only be used by at most one Program Invocation.

num_of_capab — This is the number of elements in the **capab_list** array.

capab_list — This array of pointers to implementation-specific null-terminated character strings describes limitations on the resources and capabilities of the VMD that are to be a part of the downloaded domain.

num_of_fname — These are the number of elements in the file name sequence. They specify the name of the file from which the domain is to be downloaded.

`fname_list` This list of structures of type **`FILE_NAME`** contains the elements comprising the file name sequence that specifies the name of the file from which the domain is to be downloaded. See **Volume 3 — Module 9 — File Access and Management** for more information on the structure of MMS file names.

NOTE: FOR REQUEST ONLY, when allocating a data structure of type **`RDDWN_REQ_INFO`**, enough memory must be allocated to hold the pointers for the capabilities list contained in **`capab_list`** and the file name contained in **`fname_list`**. The following C statement can be used:

```
info =(RDDWN_REQ_INFO*) chk_malloc (sizeof (RDDWN_REQ_INFO) + (num_capab *
       (sizeof(ST_CHAR *))) + (num_of_fname * sizeof(FILE_NAME)));
```

Paired Primitive Interface Functions

mp_rddwn

Usage: This primitive request function sends a RequestDomainDownload request PDU to a remote node. It uses the data found in a structure of type **RDDWN_REQ_INFO** pointed to by **info**. This is used to request that a remote node initiate a download sequence to the specified domain, contained on the local node, from the specified file on the remote node. This service may be used to respond to a LoadDomainContent indication that specified the use of a third party. See page 2-281 for more information on **mp_loaddom**. The node responding to the RequestDomainDownload should first initiate the download, respond to the DownloadSegment requests to download the domain, and respond to the TerminateDownloadSequence request before to responding positively to this request.

If an error occurs during the downloading of the domain, there is an error response sent to this service request. This should be the same error that would have been received or sent to the download service requests used to download the domain. At some point, a RequestDomainDownload service request may become uncancelable because of the complexity involved in canceling the download.

Function Prototype:

```
MMSREQ_PEND *mp_rddwn (ST_INT chan,
                       RDDWN_REQ_INFO *info);
```

Parameters:

`chan` This is the channel number over which the PDU is to be sent.

`info` This pointer to an Operation-Specific data structure of type **RDDWN_REQ_INFO** contains data specific to the request PDU to be sent.

Return Value:

`MMSREQ_PEND *` This pointer to the request control data structure of type **MMSREQ_PEND** is used to send the PDU. In case of an error, the pointer is set to null and **mms_op_err** is written with the error code.

Corresponding User Confirmation Function: **u_mp_rddwn_conf**

Operation-Specific Data Structure Used: **RDDWN_REQ_INFO**

See page 2-339 for a detailed description of this structure.

u_rddwn_ind

Usage: This user indication function is called when a RequestDomainDownload indication is received by a Server node. The contents and operation of this function are user-defined and depend on the application requirements. To conform to the MMS protocol requirements, the application must do one of the following some time after the indication is received:

1) send a positive response using the primitive response function (**mp_rddwn_resp**) after the specified domain has been downloaded from the specified file, or

2) call the appropriate primitive negative (error) response function (**mp_err_resp**). See **Volume 3 — Module 11 — MMS-EASE Error Handling** for more information on this function.

See **Volume 1— Module 1 — User Indication Function Class** for additional information on the recommended handling of indications.

Function Prototype:

```
ST_VOID u_rddwn_ind (MMSREQ_IND *ind);
```

Parameters:

`ind` This pointer to the **MMSREQ_IND** structure is described in **Volume 1 — Module 2 — Request and Indication Control Data Structures**. This pointer must be used to send the response to this indication. The primary item of interest in this structure is **ind->req_info_ptr**. This is a pointer to the operation-specific data structure for the RequestDomainDownload indication (**RDDWN_REQ_INFO**). This pointer will always be valid when **u_rddwn_ind** is called.

Return Value: `ST_VOID` (ignored)

Operation-Specific Data Structure Used: **RDDWN_REQ_INFO**

See page 2-339 for a detailed description of this structure.

mp_rddwn_resp

Usage: This primitive response function sends a RequestDomainDownload positive response PDU. This function should be called as a response to the **u_rddwn_ind** function being called (a RequestDomainDownload indication was received), and after the specified domain has been downloaded from the specified file. To respond to the RequestDomainDownload indication, your program should:

1. initiate the download sequence,
2. download the domain by responding to the multiple DownloadSegment indications, and
3. respond to the terminate download sequence indication before responding positively to the RequestDomainDownload request using this function.

If an error occurs during the downloading of the domain, there is an error response sent to the RequestDomainDownload. This should be the same error that would have been received or sent to the download service requests used to download the domain. At some point, a RequestDomainDownload service request may become uncancelable because of the complexity involved in canceling the download due to the third party operations included.

Function Prototype: `ST_RET mp_rddwn_resp (MMSREQ_IND *ind);`

Parameters:

`ind` This pointer to the indication control structure of type **MMSREQ_IND** is passed to the **u_rddwn_ind** function when it was called.

Return Value: `ST_RET` **SD_SUCCESS**. No Error.

`<> 0` Error Code.

Corresponding User Indication Function: **u_rddwn_ind**

Operation-Specific Data Structure Used: NONE

u_mp_rddwn_conf

Usage: This primitive user confirmation function is called when a confirm to a RequestDomainDownload request (**mp_rddwn**) is received. **resp_err** contains a value indicating whether an error occurred. **resp_info_ptr** may contain data that the remote node sent back in the confirmation or it may contain error information if an error occurred. See **Volume 1 — Module 1 — Primitive User Confirmation Function Class** for more information on the recommended handling of confirms.

Note that the function called for this event may be changed if desired; however, **u_mp_rddwn_conf** is the standard default function.

Function Prototype: `ST_VOID u_mp_rddwn_conf (MMSREQ_PEND *req);`

Parameters:

`req` This pointer to the **MMSREQ_PEND** structure is returned from the original **mp_rddwn** request function. See **Volume 1 — Module 2 — Request and Indication Control Data Structures** for more information on this structure.

For a positive response (**req->resp_err = CNF_RESP_OK**), the request control structure member **resp_info_pres** indicates that there is no confirm data for this function.

For a negative response, or other errors, see **Volume 3 — Module 11 — MMS-EASE Error Handling** for more information.

Return Value: `ST_VOID` (ignored)

Operation-Specific Data Structure Used: NONE

13. RequestDomainUpload Service

This service is used by a VMD to request that the Client perform an upload of a domain in the VMD.

Primitive Level RequestDomainUpload Operations

The following section contains information on how to use the paired primitive interface for the RequestDomainUpload service. It covers available data structures used by the PPI, and the four primitive level functions that together make up the RequestDomainUpload service. See **Volume 1 — Module 2 — General Sequence of Events — Confirmed Services** for a detailed explanation of the sequence of events that occur within the MMS-EASE environment during confirmed services.

- The RequestDomainUpload service consists of the paired primitive functions of **mp_rdupl**, **u_rdupl_ind**, **mp_rdupl_resp**, and **u_mp_rdupl_conf**.

Data Structures

Request/Indication

The operation specific structure described below is used by the Client in issuing a RequestDomainUpload request (**mp_rdupl**). It is received by the Server when a RequestDomainUpload indication (**u_rdupl_ind**) is received.

```
struct rdupl_req_info
  {
  ST_CHAR   dname [MAX_IDENT_LEN +1];
  ST_INT    num_of_fname;
/*FILE_NAME fname_list[ num_of_fname ];             */
  SD_END_STRUCT
  };
typedef struct rdupl_req_info RDUPL_REQ_INFO;
```

Fields:

`dname` This contains the name of the local domain to be uploaded.

`num_of_fname` This indicates the number of elements in the file name sequence. It specifies the name of the file in which the uploaded domain data is to be stored.

`fname_list` This list of structures of type **FILE_NAME** contains the elements comprising the file name sequence that specifies the name of the file in which the uploaded domain data is to be stored.

NOTES:

1. See the description of the file name structure in **Volume 3 — Module 9 — File Access and Management** for more information on the format of MMS file names.
2. FOR REQUEST ONLY, when allocating a data structure of type **RDUPL_REQ_INFO**, enough memory must be allocated to hold the information for the file name sequence. The following C statement can be used:

```
info = (RDUPL_REQ_INFO *) chk_malloc (sizeof (RDUPL_REQ_INFO) + num_of_fname *
            (sizeof (FILE_NAME));
```

Paired Primitive Interface Functions

mp_rdupl

Usage: This primitive request function sends a RequestDomainUpload request PDU to a remote node. It uses the data found in a structure of type **RDUPL_REQ_INFO,** pointed to by **info**. This is used to request that a remote node upload the specified local domain to the specified file on the remote node. This service may be used to respond to a StoreDomainContent indication that specifies the use of a third party. See page 2-287 for more information on **mp_storedom**. To respond to this request, the remote node should first initiate the upload sequence, upload the specified domain, and terminate the upload sequence. It then should store it in the specified file before responding to this request.

If an error occurs during the uploading of the domain, an error response is sent to this service request. This should be the same error that would have been received to the upload service requests used to upload the domain. At some point, a RequestDomainUpload service request may become uncancelable because of the complexity involved in canceling the upload.

Function Prototype:

```
MMSREQ_PEND *mp_rdupl (ST_INT chan,                    RDUPL_REQ_INFO *info);
```

Parameters:

chan	This is the channel number over which the RequestDomainUpload PDU is to be sent.
info	This pointer to an Operation-Specific data structure **RDUPL_REQ_INFO** contains data specific to the request PDU to be sent.

Return Value:

MMSREQ_PEND *	This pointer to the request control data structure of type **MMSREQ_PEND** is used to send the PDU. In case of an error, the pointer is set to null and **mms_op_err** is written with the error code.

Corresponding User Confirmation Function: **u_mp_rdupl_conf**

Operation-Specific Data Structure Used: **RDUPL_REQ_INFO**

See page 2-345 for a detailed description of this structure.

u_rdupl_ind

Usage: This user indication function is called when a RequestDomainUpload indication is received by a Server node. The contents and operation of this function are user-defined and depend on the application requirements. To conform to the MMS protocol requirements, the application must do one of the following some time after the indication is received:

1) send a positive response using the primitive response function (**mp_rdupl_resp**) after the specified domain has been uploaded from the specified file, or

2) call the appropriate primitive negative (error) response function (**mp_err_resp**). See **Volume 3 — Module 11 — MMS-EASE Error Handling** for more information on this function.

See **Volume 1 — Module 1 — User Indication Function Class** for additional information on the recommended handling of indications.

Function Prototype: `ST_VOID u_rdupl_ind (MMSREQ_IND *ind);`

Parameters:

`ind` This pointer to the **MMSREQ_IND** structure is described in **Volume 1 — Module 2— Request and Indication Control Data Structures**. This pointer must be used to send the response to this indication. The primary item of interest in this structure is **req_ptr->req_info_ptr**. This is a pointer to the operation-specific data structure for the RequestDomainUpload indication (**RDUPL_REQ_INFO**). This pointer will always be valid when **u_rdupl_ind** is called.

Return Value: `ST_VOID` (ignored)

Operation-Specific Data Structure Used: **RDUPL_REQ_INFO**

See page 2-345 for a detailed description of this structure.

mp_rdupl_resp

Usage: This primitive response function sends a RequestDomainUpload positive response PDU. This function should be called as a response to the **u_rdupl_ind** function being called (a RequestDomainUpload indication was received), and after the specified domain has been successfully uploaded to the specified file. To respond to the RequestDomainUpload indication your program should:

1. initiate the upload sequence,
2. upload the domain, and
3. terminate the upload sequence before responding positively to the RequestDomainUpload request using this function.

If an error occurs during the uploading of the domain, an error response is sent to the RequestDomainUpload request. This should be the same error that would have been received to the upload service requests used to upload the domain. At some point, a RequestDomainUpload service request may become uncancelable because of the complexity involved in canceling the upload in progress due to the third party operations.

Function Prototype: `ST_RET mp_rdupl_resp (MMSREQ_IND *ind);`

Parameters:

`ind` This pointer to the indication control structure of type **MMSREQ_IND** is passed to the **u_rdupl_ind** function when it was called.

Return Value: `ST_RET` **SD_SUCCESS**. No Error.

`<> 0` Error Code.

Corresponding User Indication Function: **u_rdupl_ind**

Operation-Specific Data Structure Used: NONE

u_mp_rdupl_conf

Usage: This primitive user confirmation function is called when a confirm to a RequestDomain-Upload request (**mp_rdupl**) is received. **resp_err** contains a value indicating whether an error occurred. **resp_info_ptr** may contain data that the remote node sent back in the confirmation or it may contain error information if an error occurred. See **Volume 1 — Module 1 — Primitive User Confirmation Function Class** for more information on the recommended handling of confirms.

Note that the function called for this event may be changed if desired; however, **u_mp_rdupl_conf** is the standard default function.

Function Prototype:

```
ST_VOID u_mp_rdupl_conf (MMSREQ_PEND *req);
```

Parameters:

`req` This pointer to the **MMSREQ_PEND** structure is returned from the original **mp_rdupl** request function. See **Volume 1 — Module 2 — Request and Indication Control Data Structures** for more information on this structure.

For a positive response (**req->resp_err = CNF_RESP_OK**), the request control structure member **resp_info_pres** indicates that there is no confirm data for this function.

For a negative response, or other errors, see **Volume 3 — Module 11 — MMS-EASE Error Handling** for more information.

Return Value: `ST_VOID` (ignored)

Operation-Specific Data Structure Used: NONE

14. LoadDomainContent Service

This service is used to tell a VMD (Server) to download (load) a domain from a file. This file may be local to the VMD or may be contained on an external file server.

Primitive Level LoadDomainContent Operations

The following section contains information on how to use the paired primitive interface for the LoadDomain-Content service. It covers available data structures used by the PPI, and the four primitive level functions that together make up the LoadDomainContent service. See **Volume 1 — Module 2 — General Sequence of Events — Confirmed Services** for a detailed explanation of the sequence of events that occur within the MMS-EASE environment during confirmed services.

- The LoadDomainContent service consists of the paired primitive functions of **mp_loaddom**, **u_loaddom_ind**, **mp_loaddom_resp**, and **u_mp_loaddom_conf**.

Data Structures

Request/Indication

The operation specific structure described below is used by the Client in issuing a LoadDomainContent request (**mp_loaddom**). It is received by the Server when a LoadDomainContent indication (**u_loaddom_ind**) is received.

```
struct loaddom_req_info
  {
  ST_CHAR    dname [MAX_IDENT_LEN +1];
  ST_BOOLEAN sharable;
  ST_BOOLEAN third_pty_pres;
  ST_INT     third_pty_len;
  ST_UCHAR   *third_pty;
  ST_INT     num_of_capab;
  ST_INT     num_of_fname;
/*ST_CHAR    *capab_list [num_of_capab];                                  */
/*FILE_NAME  fname_list[ num_of_fname ];                                  */
  SD_END_STRUCT
  };
typedef struct loaddom_req_info  LOADDOM_REQ_INFO;
```

Fields:

`dname` This contains the name of the Domain to be loaded.

`sharable` **SD_TRUE**. After loading the specified domain, it may be used by more than one Program Invocation concurrently.

SD_FALSE. After loading the domain specified, it may only be used by at most one Program Invocation.

`third_pty_pres` **SD_TRUE**. Include the third party AP title (**third_pty**) in the PDU. This means the file to be downloaded must be obtained from the entity identified by **third_pty**.

SD_FALSE. Do not include **third_pty** in the PDU. This means either the file to be downloaded is on the node being downloaded, or that the node knows the location of the file and can retrieve it.

`third_pty_len` This is the length, in bytes, of the data pointed to by **third_pty**.

`third_pty` This pointer to an ASN.1-encoded explicit Application Reference corresponds to the application. This contains the file to be downloaded into the specified domain. It is used to specify the system that owns the file, if it is not on the same node as the domain, or if it is not on a node that the responder knows about. See **Volume 3 — Module 10 — Third Party Handling** for more information on Application References.

`num_of_capab` This indicates the number of elements in the **`capab_list`** array.

`capab_list` This array of pointers to implementation specific null-terminated character strings describes the limitations on the resources and capabilities of the VMD that are to be a part of the domain to be loaded.

`num_of_fname` This indicates the number of elements in the file name sequence. It specifies the name of the file to be downloaded.

`fname_list` This list of structures of type **`FILE_NAME`** contains the elements comprising the file name sequence that specifies the name of the file to be downloaded. See **Volume 3 — Module 9 — File Access and Management** for more information on the structure of MMS file names.

NOTE: FOR REQUESTS ONLY, when allocating a data structure of type **`LOADDOM_REQ_INFO`**, enough memory must be allocated to hold the information for the capabilities list contained in **`capab_list`** and the file name contained in **`fname_list`**. The following C statement can be used:

```
info = (LOADDOM_REQ_INFO *) chk_malloc (sizeof (LOADDOM_REQ_INFO) + (num_capab *
            (sizeof (ST_CHAR *))) + (num_of_fname * sizeof(FILE_NAME)));
```

Paired Primitive Interface Functions

mp_loaddom

Usage: This primitive request function sends a LoadDomainContent request PDU using the data from a structure of type **LOADDOM_REQ_INFO**, pointed to by **info**. The LoadDomainContent service is used to request the remote node to load the specified domain from a file in either its own filestore or another subordinate (third party) system's filestore. If the file is on a third party system, the remote node should use the RequestDomainDownload services to download its domain before responding to this request. See page 2-341 for more information on **mp_rddwn**. Any error response to a LoadDomainContent request specifying a third party should be the error sent back in the response to the RequestDomainDownload request.

Function Prototype:

```
MMSREQ_PEND *mp_loaddom (ST_INT chan,
                         LOADDOM_REQ_INFO *info);
```

Parameters:

`chan` This integer contains the channel number over which the LoadDomainContent PDU is to be sent.

`info` This pointer to an Operation-Specific data structure of type **LOADDOM_REQ_INFO** contains information specific to the request PDU to be sent.

Return Value:

`MMSREQ_PEND *` This pointer to the request control data structure of type **MMSREQ_PEND** is used to send the PDU. In case of an error, the pointer is set to null and **mms_op_err** is written with the error code.

Corresponding User Confirmation Function: **u_mp_loaddom_conf**

Operation-Specific Data Structure Used: **LOADDOM_REQ_INFO**

See page 2-351 for a detailed description of this structure.

u_loaddom_ind

Usage: This user indication function is called when a LoadDomainContent indication is received by a Server node. The contents and operation of this function are user-defined and depend on the application requirements. To conform to the MMS protocol requirements, the application must do one of the following some time after the indication is received:

1) send a positive response using the primitive response function (**mp_loaddom_resp**) after the specified domain was successfully loaded from the specified file, or

2) call the appropriate primitive negative (error) response function (**mp_err_resp**). See **Volume 3 — Module 11 — MMS-EASE Error Handling** for more information on this function.

See **Volume 1 — Module 1 — User Indication Function Class** for additional information on the recommended handling of indications.

Function Prototype:

```
ST_VOID u_loaddom_ind (MMSREQ_IND *ind);
```

Parameters:

`ind` This pointer to the **MMSREQ_IND** structure is described in **Volume 1 — Module 2 — Request and Indication Control Data Structures**. This pointer must be used to send the response to this indication. The primary item of interest in this structure is **ind->req_info_ptr**. This is a pointer to the operation-specific data structure for the Load Domain Content indication (**LOADDOM_REQ_INFO**). This pointer will always be valid when **u_loaddom_ind** is called.

Return Value: `ST_VOID` (ignored)

Operation-Specific Data Structure Used: **LOADDOM_REQ_INFO**

See page 2-351 for a detailed description of this structure.

mp_loaddom_resp

Usage: This primitive response function sends a LoadDomainContent positive response PDU. This function should be called as a response to the **u_loaddom_ind** function being called (a LoadDomainContent indication was received), and after the specified domain was successfully loaded from the specified file. This service is used to have an external node request that a domain, contained in your VMD, be loaded from the specified file. This file may be on a third party node identified in the indication. If the file is in local filestore, the response should be sent after the domain is downloaded. If a third party must be used, your application program should first issue a RequestDomainDownload request (using **mp_rddwn**) to the node that has the specified file in order to have domain downloaded.

The LoadDomainContent indication can be responded to positively using this function. This can occur only after the download is completed successfully, and a positive response to the RequestDomainDownload is received from the third party node to the RequestDomainDownload request sent. If an error occurs during the download of the domain, this same error code should be included in any error response sent to the LoadDomainContent indication. A LoadDomainContent service request may not be cancelable because of the complexity involved in canceling the download sequence due to the third party operations.

Function Prototype: `ST_RET mp_loaddom_resp (MMSREQ_IND *ind);`

Parameters:

`ind` This pointer to the indication control structure of type **MMSREQ_IND** is passed to the **u_loaddom_ind** function when it was called.

Return Value: `ST_RET` **SD_SUCCESS**. No Error.

`<> 0` Error Code.

Corresponding User Indication Function: **u_loaddom_ind**

Operation-Specific Data Structure Used: NONE

u_mp_loaddom_conf

Usage: This primitive user confirmation function is called when a confirm to a LoadDomainContent request (**mp_loaddom**) is received. **resp_err** contains a value indicating whether an error occurred. **resp_info_ptr** may contain data that the remote node sent back in the confirmation or it may contain error information if an error occurred. See **Volume 1 — Module 1 — Primitive User Confirmation Function Class** for more information on the recommended handling of confirms.

Note that the function called for this event may be changed if desired; however, **u_mp_loaddom_conf** is the standard default function.

Function Prototype:

```
ST_VOID u_mp_loaddom_conf (MMSREQ_PEND *req);
```

Parameters:

`req` This pointer to the **MMSREQ_PEND** structure is returned from the original **mp_loaddom** request function. See **Volume 1— Module 2 — Request and Indication Control Data Structures** for more information on this structure.

For a positive response (**req->resp_err = CNF_RESP_OK**), the request control structure member **resp_info_pres** indicates that there is no confirm data for this function.

For a negative response, or other errors, see **Volume 3 — Module 11 — MMS-EASE Error Handling** for more information.

Return Value: `ST_VOID` (ignored)

Operation-Specific Data Structure Used: NONE

15. StoreDomainContent Service

This service is used to request that the contents of a specified domain at the Server be stored in a file on some filestore. This requires additional processing so that this domain can be loaded later using the LoadDomain-Content service.

Primitive Level StoreDomainContent Operations

The following section contains information on how to use the paired primitive interface for the StoreDomain-Content service. It covers available data structures used by the PPI, and the four primitive level functions that together make up the StoreDomainContent service. See **Volume 1— Module 2 — General Sequence of Events — Confirmed Services** for a detailed explanation of the sequence of events that occur within the MMS-EASE environment during confirmed services.

- The StoreDomainContent service consists of the paired primitive functions of **`mp_storedom`**, **`u_storedom_ind`**, **`mp_storedom_resp`**, and **`u_mp_storedom_conf`**.

Data Structures

Request/Indication

The operation specific structure described below is used by the Client in issuing a StoreDomainContent request (**`mp_storedom`**). It is received by the Server when a StoreDomainContent indication (**`u_storedom_ind`**) is received.

```
struct storedom_req_info
  {
  ST_CHAR    dname [MAX_IDENT_LEN +1];
  ST_INT     num_of_fname;
  ST_BOOLEAN third_pty_pres;
  ST_INT     third_pty_len;
  ST_UCHAR   *third_pty;
/*FILE_NAME  fname_list[ num_of_fname ];                          */
  SD_END_STRUCT
  };
typedef struct storedom_req_info STOREDOM_REQ_INFO;
```

Fields:

`dname` This contains the name of the domain to be stored into the specified file.

`third_pty_pres` **`SD_TRUE`**. Include the third party AP title (**`third_pty`**) in the PDU. This means the file in which the domain is to be stored is located on the external entity identified by **`third_pty`**.

`SD_FALSE`. Do not include **`third_pty`** in the PDU. This means that the file where the domain is to be stored to is either on the same node that the domain is located or that the VMD knows where to place the file.

`third_pty_len` This is the length, in bytes, of the data pointed to by **`third_pty`**.

`third_pty` This pointer to an ASN.1-encoded explicit Application Reference corresponds to the application containing the file that will store the domain. It is used when the file that will contain this domain is not on the same system as the domain. See **Volume 3 — Module 10 — Third Party Handling** for an explanation of third party references.

`num_of_fname` This indicates the number of elements in file name sequence. It specifies the file name into which to store the domain.

`fname_list` This list of structures of type **`FILE_NAME`** contains the elements that comprise the file name sequence specifying the file name into which to store the domain. See **Volume 3 — Module 9 — File Access and Management** for more information on the structure of MMS file names.

NOTE: FOR REQUEST ONLY, when allocating a data structure of type **`STOREDOM_REQ_INFO`**, you also must allocate enough memory to hold the file name contained in **`fname_list`**. The following C statement can be used:

```
info = (STOREDOM_REQ_INFO *) chk_malloc (sizeof (STOREDOM_REQ_INFO) +
            (num_of_fname * sizeof (FILE_NAME)));
```

Paired Primitive Interface Functions

mp_storedom

Usage: This primitive request function sends a StoreDomainContent request PDU using the data from a structure of type **STOREDOM_REQ_INFO**, pointed to by **info**. The StoreDomainContent service is used to request that the contents of a domain on the remote node be stored in a file. This file may be on the remote node, or another third party file system. If the file is on a third party system, the remote node should use the RequestDomainUpload services to upload the domain before responding to this request. See page 2-346 for more information on **mp_rdupl**. Any error response to a StoreDomainContent request specifying a third party should be the error sent back in the response to the RequestDomainUpload request.

Function Prototype:

```
MMSREQ_PEND *mp_storedom (ST_INT chan,
                          STOREDOM_REQ_INFO *info);
```

Parameters:

chan	This integer contains the channel number over which the StoreDomainContent PDU is to be sent.
info	This pointer to an Operation-Specific data structure of type **STOREDOM_REQ_INFO** contains information specific to the request PDU to be sent.

Return Value:

MMSREQ_PEND *	This pointer to the request control structure of type **MMSREQ_PEND** is used to send the PDU. In case of an error, the pointer is set to null and **mms_op_err** is written with the error code.

Corresponding User Confirmation Function: **u_mp_storedom_conf**

Operation-Specific Data Structure Used: **STOREDOM_REQ_INFO**

See page 2-357 for a detailed description of this structure.

u_storedom_ind

Usage: This user indication function is called when a StoreDomainContent indication is received by a Server node. The contents and operation of this function are user-defined and depend on the application requirements. To conform to the MMS protocol requirements, the application must do one of the following some time after the indication is received:

1) send a positive response using the primitive response function (**mp_storedom_resp**) after the specified domain has been successfully stored to the specified file, or

2) call the appropriate primitive negative (error) response function (**mp_err_resp**). See **Volume 3 — Module 11 — MMS-EASE Error Handling** for more information on this function.

See **Volume 1 — Module 1 — User Indication Function Class** for additional information on the recommended handling of indications.

Function Prototype:

```
ST_VOID u_storedom_ind (MMSREQ_IND *ind);
```

Parameters:

`ind` This pointer to the **MMSREQ_IND** structure is described in **Volume 1 — Module 1 — Request and Indication Control Data Structures**. This pointer must be used to send the response to this indication. The primary item of interest in this structure is **ind->req_info_ptr**. This is a pointer to the operation-specific data structure for a StoreDomainContent indication (**STOREDOM_REQ_INFO**). This pointer is always valid when **u_storedom_ind** is called.

Return Value: `ST_VOID` (ignored)

Operation-Specific Data Structure Used: **STOREDOM_REQ_INFO**

See page 2-357 for a detailed description of this structure.

mp_storedom_resp

Usage: This primitive response function sends a StoreDomainContent positive response PDU. This function should be called as a response to the **u_storedom_ind** function being called (a StoreDomainContent indication was received), and after the specified domain has been successfully stored to the specified file. This service is used to have an external node request that a domain, contained in your VMD, is uploaded to the specified file. This file may be on a third party node identified in the indication. If the file is in the local filestore, the response should be sent after the domain has been uploaded. If a third party must be used, your program should first issue a RequestDomainUpload request (using **mp_rdupl**) to the node that has the specified file in order to upload the domain to the file.

The StoreDomainContent indication can be responded to positively using this function. This can occur after the upload is complete, and you have responded positively to the third party's request to terminate the upload sequence. If an error occurs during the upload of the domain, this same error code should be included in any error response sent to the StoreDomainContent indication. A StoreDomainContent service request may not be cancelable because of the complexity involved in canceling the upload sequence due to the third party operations.

Function Prototype: `ST_RET mp_storedom_resp (MMSREQ_IND *ind);`

Parameters:

`ind` This pointer to the indication control structure of type **MMSREQ_IND** is passed to the **u_storedom_ind** function when it was called.

Return Value: `ST_RET` **SD_SUCCESS**. No Error.

`<> 0` Error Code.

Corresponding User Indication Function: **u_storedom_ind**

Operation-Specific Data Structure Used: NONE

u_mp_storedom_conf

Usage: This primitive user confirmation function is called when a confirm to a StoreDomainContent request (**mp_storedom**) is received. **resp_err** contains a value indicating whether an error occurred. **resp_info_ptr** may contain data that the remote node sent back in the confirmation or it may contain error information if an error occurred. See **Volume 1 — Module 1 — Primitive User Confirmation Function Class** for more information on the recommended handling of confirms.

Note that the function called for this event may be changed if desired; however, **u_mp_storedom_conf** is the standard default function.

Function Prototype:

```
ST_VOID u_mp_storedom_conf (MMSREQ_PEND *req);
```

Parameters:

`req` This pointer to the **MMSREQ_PEND** structure is returned from the original **mp_storedom** request function. See **Volume 1 — Module 2 — Request and Indication Control Data Structures** for more information on this structure.

For a positive response (**req->resp_err = CNF_RESP_OK**), the request control structure member **resp_info_pres** indicates that there is no confirm data for this function.

For a negative response, or other errors, see **Volume 3 — Module 11 — MMS-EASE Error Handling** for more information.

Return Value: `ST_VOID` (ignored)

Operation-Specific Data Structure Used: NONE

16. DeleteDomain Service

This service is used to request that a specified domain be deleted at the Server.

Primitive Level DeleteDomain Operations

The following section contains information on how to use the paired primitive interface for the DeleteDomain service. It covers available data structures used by the PPI, and the four primitive level functions that together make up the DeleteDomain service. See **Volume 1 — Module 2 — General Sequence of Events — Confirmed Services** for a detailed explanation of the sequence of events that occur within the MMS-EASE environment during confirmed services.

- The DeleteDomain service consists of the paired primitive functions of **`mp_deldom`**, **`u_deldom_ind`**, **`mp_deldom_resp`**, and **`u_mp_deldom_conf`**.

Data Structures

Request/Indication

The operation specific structure described below is used by the Client in issuing a DeleteDomain request (**`mp_deldom`**). It is received by the Server when a DeleteDomain indication (**`u_deldom_ind`**) is received.

```
struct deldom_req_info
  {
  ST_CHAR dname [MAX_IDENT_LEN +1];
  };
typedef struct deldom_req_info DELDOM_REQ_INFO;
```

Fields:

`dname` This contains the name of the domain to be deleted.

Paired Primitive Interface Functions

mp_deldom

Usage: This primitive request function sends a DeleteDomain request PDU to a remote node. It uses the data found in a structure of type **DELDOM_REQ_INFO** pointed to by **info**. The DeleteDomain service is used to request that the remote node delete the specified domain.

Function Prototype:

```
MMSREQ_PEND *mp_deldom (ST_INT chan,
                        DELDOM_REQ_INFO *info);
```

Parameters:

chan	This is the channel number over which the DeleteDomain request is to be sent.
info	This pointer to an Operation-Specific data structure of type **DELDOM_REQ_INFO** contains information specific to the request PDU to be sent.

Return Value:

MMSREQ_PEND *	This pointer to the request control data structure of type **MMSREQ_PEND** is used to send the PDU. In case of an error, the pointer is set to null and **mms_op_err** is written with the error code.

Corresponding User Confirmation Function: **u_mp_deldom_conf**

Operation-Specific Data Structure Used: **DELDOM_REQ_INFO**

See page 2-363 for a detailed description of this structure.

u_deldom_ind

Usage: This user indication function is called when a DeleteDomain indication is received by a Server node. The contents and operation of this function are user-defined and depend on the application requirements. To conform to the MMS protocol requirements, the application must do one of the following some time after the indication is received:

1) send a positive response using the primitive response function (**mp_deldom_resp**) after the specified domain has been successfully deleted, or

2) call the appropriate primitive negative (error) response function (**mp_err_resp**). See **Volume 3 — Module 11 — MMS-EASE Error Handling** for more information on this function.

See **Volume 1 — Module 1 — User Indication Function Class** for additional information on the recommended handling of indications.

Function Prototype: `ST_VOID u_deldom_ind (MMSREQ_IND *ind);`

Parameters:

`ind` This pointer to the **MMSREQ_IND** structure is described in **Volume 1 — Module 2 — Request and Indication Control Data Structures**. This pointer must be used to send the response to this indication. The primary item of interest in this structure is **ind->req_info_ptr**. This is a pointer to the operation-specific data structure for a DeleteDomain indication (**DELDOM_REQ_INFO**). This pointer will always be valid when **u_deldom_ind** is called.

Return Value: `ST_VOID` (ignored)

Operation-Specific Data Structure Used: **DELDOM_REQ_INFO**

See page 2-363 for a detailed description of this structure.

mp_deldom_resp

Usage: This primitive response function sends a DeleteDomain positive response PDU. This function should be called after the **u_deldom_ind** function is called (a DeleteDomain indication was received), and after the specified domain has been successfully deleted. You should only allow domains to be deleted that are not currently being uploaded, are in the READY state (not being used by a program invocation), and have the MMS deletable attribute set. When deleting a domain, any domain specific objects also should be deleted. This includes such objects as variables, types, semaphores, and event conditions associated with the domain to be deleted.

Function Prototype:

```
ST_RET mp_deldom_resp (MMSREQ_IND *ind);
```

Parameters:

`ind` This pointer to the indication control structure of type **MMSREQ_IND** is passed to the **u_deldom_ind** function when it was called.

Return Value: `ST_RET`

SD_SUCCESS. No Error.

`<> 0` Error Code.

Corresponding User Indication Function: **u_deldom_ind**

Operation-Specific Data Structure Used: NONE

u_mp_deldom_conf

Usage: This primitive user confirmation function is called when a confirm to a DeleteDomain request (**mp_deldom**) is received. **resp_err** contains a value indicating whether an error occurred. **resp_info_ptr** may contain data that the remote node sent back in the confirmation or it may contain error information if an error occurred. See **Volume 1 — Module 1 — Primitive User Confirmation Function Class** for more information on the recommended handling of confirms.

Note that the function called for this event may be changed if desired; however, **u_mp_deldom_conf** is the standard default function.

Function Prototype:

```
ST_VOID u_mp_deldom_conf (MMSREQ_PEND *req);
```

Parameters:

`req` This pointer to the **MMSREQ_PEND** structure is returned from the original **mp_deldom** request function. See **Volume 1 — Module 2 — Request and Indication Control Data Structures** for more information on this structure.

For a positive response (**req->resp_err = CNF_RESP_OK**), the request control structure member **resp_info_pres** indicates that there is no confirm data for this function.

For a negative response, or other errors, see **Volume 3 — Module 11 — MMS-EASE Error Handling** for more information.

Return Value: `ST_VOID` (ignored)

Operation-Specific Data Structure Used: NONE

17. GetDomainAttributes Service

This service is used to request that a Server return all the attributes associated with a specified domain.

Primitive Level GetDomainAttributes Operations

The following section contains information on how to use the paired primitive interface for the GetDomainAttributes service. It covers available data structures used by the PPI, and the four primitive level functions that together make up the GetDomainAttributes service. See **Volume 1 — Module 2 — General Sequence of Events — Confirmed Services** for a detailed explanation of the sequence of events that occur within the MMS-EASE environment during confirmed services.

- The GetDomainAttributes service consists of the paired primitive functions of **`mp_getdom`**, **`u_getdom_ind`**, **`mp_getdom_resp`**, and **`u_mp_getdom_conf`**.

Data Structures

Request/Indication

The operation specific structure described below is used by the Client in issuing a GetDomainAttributes request (**`mp_getdom`**). It is received by the Server when a GetDomainAttributes indication (**`u_getdom_ind`**) is received.

```
struct getdom_req_info
  {
  ST_CHAR dname [MAX_IDENT_LEN +1];
  };
typedef struct getdom_req_info GETDOM_REQ_INFO;
```

Fields:

`dname` This contains the name of the domain for which the attributes are being requested.

Response/Confirm

This operation specific data structure described below is used by the Server in issuing a GetDomainAttributes response (**`mp_getdom_resp`**). It is received by the Client when a GetDomainAttributes confirm (**`u_mp_getdom_conf`**) is received.

```
struct getdom_resp_info
  {
  ST_INT      num_of_capab;
  ST_BOOLEAN mms_deletable;
  ST_BOOLEAN sharable;
  ST_INT      num_of_pinames;
  ST_INT16    state;
  ST_BOOLEAN upload_in_progress;
/*ST_CHAR     *capab_list [num_of_capab];                                  */
/*ST_CHAR     *pinames_list [num_of_pinames];                      */
  SD_END_STRUCT
  };
typedef struct getdom_resp_info  GETDOM_RESP_INFO;
```

Fields:

`num_of_capab` This indicates the number of pointers in the capabilities list **`capab_list`**.

`mms_deletable` **`SD_FALSE`**. Domain is not deletable using a MMS service request.

`SD_TRUE`. Domain is deletable using a MMS service request.

`sharable` **`SD_TRUE`**. Domain is sharable among multiple program invocations.

`SD_FALSE`. Domain is not sharable

`num_of_pinames` This indicates the number of pointers in the program invocation list, **`pinames_list`**.

`state` This indicates the state of the Domain:

`DOM_NON_EXISTENT`. This state represents the domain before its creation.

`DOM_LOADING`. This state represents an intermediate state that occurs during the loading process.

`DOM_READY`. This state represents the state a domain enters in after a successful download.

`DOM_IN_USE`. This state differs from the Ready state in that one or more Program Invocations have been defined using this domain.

`DOM_COMPLETE`. This state represents an intermediate state that occurs after the last DownloadSegment has been received but before the DownloadSequence has been terminated.

`DOM_INCOMPLETE`. This state represents an intermediate state that when A DownloadSequence was terminated before the loading process was complete.

`DOM_D1 - DOM_D9`. These states (D1 - D9) represent intermediate states per the IS specification. These are states between a request and a response.

`upload_in_progress` This indicates the number of uploads currently in progress.

`capab_list` This array of pointers to the list of capabilities contains information about the capabilities and the VMD resource limitations of this domain.

`pinames_list` This is an array of pointers to a list of the names of the program invocations that reference this domain.

NOTE: FOR RESPONSE ONLY, when allocating a data structure of type **`GETDOM_RESP_INFO`**, enough memory must be allocated to hold the information for list of capabilities, **`capab_list`**, and the list of the program invocation names, **`pinames_list`**, contained in this structure. The following C language statement can be used:

```
info = (GETDOM_RESP_INFO *) chk_malloc (sizeof (GETDOM_RESP_INFO) +
            ((num_of_capab + num_of_pinames) * (sizeof(ST_CHAR *))));
```

Paired Primitive Interface Functions

mp_getdom

Usage: This primitive request function sends a GetDomainAttributes request PDU using the data from a structure of type **GETDOM_REQ_INFO**, pointed to by **info**. This service is used to obtain a list of the attributes of a domain at the remote node.

Function Prototype:

```
MMSREQ_PEND *mp_getdom (ST_INT chan,
                        GETDOM_REQ_INFO *info);
```

Parameters:

chan	This is the channel number over which the PDU is to be sent.
info	This pointer to an Operation-Specific data structure of type **GETDOM_REQ_INFO** contains information specific to the request PDU to be sent.

Return Value:

MMSREQ_PEND *	This pointer to the request control data structure of type **MMSREQ_PEND** is used to send the PDU. In case of an error, the pointer is set to null and **mms_op_err** is written with the error code.

Corresponding User Confirmation Function: **u_mp_getdom_conf**

Operation-Specific Data Structure Used: **GETDOM_REQ_INFO**

See page 2-369 for a detailed description of this structure.

u_getdom_ind

Usage: This user indication function is called when a GetDomainAttributes indication is received by a Server node. The contents and operation of this function are user-defined and depend on the application requirements. To conform to the MMS protocol requirements, the application must do one of the following some time after the indication is received:

1) send a positive response using either:

   a) the primitive response function(**`mp_getdom_resp`**) after the Named Type's attributes have been obtained,

   b) the virtual machine response function (**`mv_getdom_resp`**) to have the virtual machine take the requested action and send the response, or

2) call the appropriate primitive negative (error) response function (**`mp_err_resp`**). See **Volume 3 — Module 11 — MMS-EASE Error Handling** for more information on this function.

See **Volume 1 — Module 1 — User Indication Function Class** for additional information on the recommended handling of indications.

Function Prototype: `ST_VOID u_getdom_ind (MMSREQ_IND *ind);`

Parameters:

`ind` This pointer to the **`MMSREQ_IND`** structure is described in **Volume 1 — Module 2 — Request and Indication Control Data Structures**. This pointer must be used to send the response to this indication. The primary item of interest in this structure is **`ind->req_info_ptr`**. This is a pointer to the operation-specific data structure for a GetDomainAttributes indication (**`GETDOM_REQ_INFO`**). This pointer will always be valid when **`u_getdom_ind`** is called.

Return Value: `ST_VOID` (ignored)

Operation-Specific Data Structure Used: **`GETDOM_REQ_INFO`**

See page 2-369 for a detailed description of this structure.

mp_getdom_resp

Usage: This primitive response function sends a GetDomainAttributes positive response PDU using the data from a structure of type **GETDOM_RESP_INFO**, pointed to by **info**. This function should be called as a response to the **u_getdom_ind** function being called (a GetDomainAttributes indication was received), and after the specified domain's attributes have been successfully obtained.

Function Prototype:

```
ST_RET mp_getdom_resp (MMSREQ_IND *ind,
                       GETDOM_RESP_INFO *info);
```

Parameters:

`ind` This pointer to the indication control structure of type **MMSREQ_IND** is passed to the **u_getdom_ind** function when it was called.

`info` This pointer to an Operation-Specific data structure of type **GETDOM_RESP_INFO** contains information specific to the response PDU to be sent.

Return Value: `ST_RET` **SD_SUCCESS**. No Error.

`<> 0` Error Code.

Corresponding User Indication Function: **u_getdom_ind**

Operation-Specific Data Structure Used: **GETDOM_RESP_INFO**

See page 2-369 for a detailed description of this structure.

u_mp_getdom_conf

Usage: This primitive user confirmation function is called when a confirm to a GetDomainAttributes request (**mp_getdom**) is received. **resp_err** contains a value indicating whether an error occurred. **resp_info_ptr** may contain data that the remote node sent back in the confirmation or it may contain error information if an error occurred. If no error has occurred (**req->resp_err = CNF_RESP_OK**), GetDomainAttributes information is available to the application using the **req->resp_info_ptr** pointer. See **Volume 1 — Module 1 — Primitive User Confirmation Function Class** for more information on the recommended handling of confirms.

Note that the function called for this event may be changed if desired; however, **u_mp_getdom_conf** is the standard default function.

Function Prototype:

```
ST_VOID u_mp_getdom_conf (MMSREQ_PEND *req);
```

Parameters:

`req` This pointer to the **MMSREQ_PEND** structure is returned from the original **mp_getdom** request function. See **Volume 1 — Module 2 — Request and Indication Control Data Structures** for more information on this structure.

For a positive response (**req->resp_err = CNF_RESP_OK**), the request control structure member **resp_info_ptr** is a pointer to the operation-specific data structure (**GETDOM_RESP_INFO**) for the GetDomainAttributes function.

For a negative response, or other errors, see **Volume 3 — Module 11 — MMS-EASE Error Handling** for more information.

Return Value: `ST_VOID` (ignored)

Operation-Specific Data Structure Used: **GETDOM_RESP_INFO**

See page 2-369 for a detailed description of this structure.

Virtual Machine GetDomainAttributes Operations

The following section contains information on how to use the virtual machine interface for the GetDomainAttributes service. It covers the virtual machine response function that makes up the GetDomainAttributes service.

- The GetDomainAttributes service consists of the virtual machine response function of **`mv_getdom_resp`**.

There are no virtual machine data structures concerning this service.

Virtual Machine Interface Function

mv_getdom_resp

Usage: This virtual machine response function allows a user to search through the MMS-EASE database and automatically send a GetDomainAttributes positive response PDU. This function will return an error if it cannot find the specified named type object. BE SURE TO SET **`mms_chan_info.objs.vmd`** TO POINT TO THE PROPER VMD STRUCTURE BEFORE CALLING THIS FUNCTION. See **Volume 1 — Module 2 — Channel Information Structure** and **Module 4 — VMD Control** starting on page 2-10 for more information.

Function Prototype: `ST_RET mv_getdom_resp (MMSREQ_IND *req_info);`

Parameters:

`ind` This pointer to the indication control data structure of type **`MMSREQ_IND`** is received for the indication service function, **`u_getdom_ind`**.

Return Value: `ST_RET` **`SD_SUCCESS`**. No Error.

`<> 0` Error Code.

Corresponding User Indication Function: **`u_getdom_ind`**

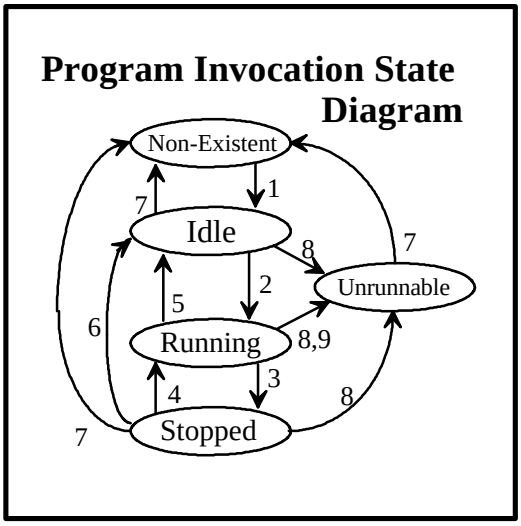
18. Program Invocation Management Introduction

This portion of MMS-EASE provides services to manage program invocations. It is through the manipulation of program invocations that a MMS client controls the execution of programs in a VMD. Program invocations can be started, stopped, reset, etc., by MMS clients. A program invocation is an execution thread that consists of a collection of one or more domains. Simple devices with simple execution structures may only support a single program invocation containing only one domain. More sophisticated devices and applications may support multiple program invocations containing several domains.

As an example, consider how the MMS execution model could be applied to a personal computer (PC). When the PC powers up, it downloads a domain called the operating system into memory. When you type the name of the program you want to run and hit the <return> key, the computer downloads another domain (the executable program) from a file. It then creates and runs a program invocation consisting of the program and the operating system. The program itself cannot be executed until it is bound to the operating system by the act of creating the program invocation. In addition to the program invocation's name, the attributes of a program invocation are:

State A program invocation is a series of operations performed in a timed sequence. In this, a program invocation undergoes a series of state changes. The figure shown below shows the relationship of the various states of a program invocation to the actual services used.

1. IDLE — This is the time before a program invocation is placed in operation.
2. RUNNING — This is the time when the program invocation is being executed.
3. STOPPED — This is the time between the beginning of execution and completion of when execution has ceased.
4. UNRUNNABLE — This is the time where the program invocation may no longer be executed, but has not yet been deleted.
5. STARTING — This is the time between the receipt of a start indication and the issuance of a start response.
6. STOPPING — This is the time between the receipt of a stop indication and the issuance of a stop response.
7. RESUMING — This is the time between the receipt of a resume indication and the issuance of a resume response.
8. RESETTING — This is the time between the receipt of a reset indication and the issuance of a reset response.



MMS Clients use MMS services to cause state transitions in the program invocation as follows:

1. *CreateProgramInvocation service request.*
2. *Start service request.*
3. *Stop service request or program stop.*
4. *Resume service request.*
5. *End of program and Reusable=TRUE.*
6. *Reset service request.*
7. *DeleteProgramInvocation service request.*
8. *Kill service request or error condition.*
9. *End of program and Reusable=FALSE.*

FIGURE 4: Program Invocation State Diagram

List of Domains The list of domains that comprise the program invocation.

Deletable Indicates if the program invocation is deletable using the DeleteProgramInvocationService.

Reusable Reusable program invocations automatically reenter the IDLE state when the program invocation arrives at the end of the program. Otherwise, program invocations in the RUNNING state must be STOPPED, then RESET to bring it back to the IDLE state.

Monitored Monitored program invocations use the MMS Event Management services to inform the MMS client when the program invocation leaves the RUNNING state. Monitored program invocations have an event condition object defined with the same name as the program invocation.

Execution Argument This is a character string passed to the program invocation using the START or RESUME service. The execution argument is used to pass data to the program invocation like parameters in a subroutine call.

There are eight services used by Program Invocation Management:

CreateProgramInvocation This service is used by a Client to create a program invocation at a MMS Server. See page 2-387 for more information.

DeleteProgramInvocation This service is used to delete a program at a MMS Server. See page 2-395 for more information.

Start This service is used to start a program invocation executing from the IDLE state at a MMS Server. See page 2-401 for more information.

Stop This service is used to stop a running program invocation at a MMS Server. See page 2-407 for more information.

Resume This service is used to restart a program invocation that has been temporarily stopped at a MMS Server. See page 2-413 for more information.

Reset This service is used to put a stopped or running program invocation back into the IDLE state at a MMS Server. See page 2-419 for more information.

Kill This service is used to put a stopped or running program invocation into an UNRUNNABLE state at a MMS Server. See page 2-425 for more information.

GetProgramInvocationAttributes This service is used by a MMS Client to determine the attributes of a program invocation at a MMS Server. See page 2-431 for more information.

19. Virtual Machine Interface

The MMS-EASE virtual machine defines variables, data structures, and functions to assist applications with managing information relating to MMS program invocations. The following section describes the data structure and variables used by the MMS-EASE virtual machine to keep track of program invocations. These structures also can be accessed and written by your application programs.

Common Program Invocation Data Structures

Program Invocation Control Structure

The following structure keeps track of program invocations. It is used to respond automatically to requests for reports or name-lists on them. Do not access any members of this structure marked with **/* DO NOT USE */**.

```
struct prog_inv
  {
  DBL_LNK        link;
  ST_CHAR        pi_name[MAX_IDENT_LEN + 1];
  ST_UCHAR       protection;
  ST_INT         state;
  ST_BOOLEAN     deletable;
  ST_BOOLEAN     reusable;
  ST_BOOLEAN     monitor;
  ST_UCHAR       *start_arg;
  ST_INT          start_len;
  ST_INT         ndom;
  ST_CHAR        reserved[4];
  ST_CHAR        user_rsrvd[8];
/*NAMED_DOM_CTRL *dom_list[];          */
  SD_END_STRUCT
  };
typedef struct prog_inv   PROG_INV;
```

Fields:

`link` **/* FOR INTERNAL USE ONLY - DO NOT USE. */**

`pi_name` This contains the name of the program invocation. This must be Visible String of no longer than 32 characters existing only of numbers (0-9), upper case letters (A-Z), lower case letters (a-z), and the **_** and **$** characters. It must not start with a number or contain spaces.

`protection` This is compared to the privilege allowed for a given remote communications entity so that access to this program invocation can be controlled.

`state` This variable contains the state of the program invocation:

PI_NON_EXISTENT. This indicates the condition of the Program Invocation before it is created.

PI_UNRUNNABLE. This indicates that the Program Invocation can no longer be executed.

PI_IDLE. This indicates the state of the Program Invocation after it has been created and before it is placed in operation.

PI_RUNNING. This indicates the Program Invocation state during its execution.

PI_STOPPED. This indicates that the Program Invocation is in an intermediate state between the beginning of an execution and its end.

PI_STARTING. This indicates that the Program Invocation is in an intermediate state between IDLE and RUNNING.

PI_STOPPING. This indicates that the Program Invocation is in an intermediate state between RUNNING and STOPPED.

PI_RESUMING. This indicates that the Program Invocation is in an intermediate state between STOPPED and RUNNING.

PI_RESETTING. This indicates that the Program Invocation is in an intermediate state between STOPPED and IDLE.

`deletable` **SD_TRUE**. This program invocation can be deleted over the network using the Delete Program Invocation service.

SD_FALSE. This program invocation cannot be deleted over the network.

`reusable` **SD_TRUE**. This program invocation can be reset and started from the beginning. (This is not the same as resetting).

SD_FALSE. This program invocation cannot be reset and started from the beginning.

`monitor` **SD_TRUE**. This virtual machine should notify the host of this program invocation's status using the Event Notification service. See **Volume 3 — Module 7 — Event Management** for more information on Events.

SD_FALSE. The virtual machine will not notify the host through the Event Notification service.

`start_arg` This points to the start argument of the last Start request issued. This is a null-terminated character string used to pass data to the program invocation.

`start_len` This is the length, in bytes, of **start_arg** including the null character.

`ndom` This indicates the number of domains present in this program invocation.

`user_rsrvd` The data in this element is user-defined, and can be used for any purpose.

`dom_list` This array of pointers points to the **NAMED_DOM_CTRL** structures of the domains referenced by this program invocation. See page 2-279 for more information on this structure. This array follows immediately after the **PROG_INV** structure (contiguous in memory).

PROG_INV *vmd_pi_ctrl;

This is a pointer to the list of program invocations contained in the specified VMD. See **Module 4** starting on page 2-10 for more information on VMDs and the VMD control structure.

Program Invocation Variables

extern ST_INT mms_pi_count;

This variable keeps a running count of the program invocations currently defined for all the defined VMDs.

extern ST_INT max_mmsease_pis;

This variable indicates the maximum number of program invocations allowed at one time in the VMD database. This is an aggregate of program invocations for ALL VMDs. The default value for this variable is 16.

NOTE: When using the Virtual Machine response functions for program invocations, caution must be made to use ALL the associated available virtual machine functions for ALL the services supplied. *DO NOT MIX VIRTUAL MACHINE FUNCTIONS WITH PAIRED PRIMITIVE FUNCTIONS*. It may confuse the program invocation state machine, and will cause unpredictable results.

Program Invocation Support Functions

These functions are called by the user application program to perform the various support functions required by MMS-EASE to manage domains. For Program Invocation Management, they are used to add, change, and delete entries into the list of program invocation structures, and check the state of a specified program invocation. The contents of these functions are determined by MMS-EASE.

ms_add_pi

Usage: This support function is used to insert a new program invocation control structure into the list of program invocations associated with a particular VMD. The VMD is selected by setting the **m_vmd_select** pointer to the appropriate VMD control structure, before to calling this function. See page 2-379 for more information on the program invocation control structure, and **Module 4 — VMD Control** on page 2-10 for information on the VMD structure.

This function allocates memory for a new program invocation control structure (**PROG_INV**), and sets the members to the following default values:

state	**PI_IDLE**
reusable	**SD_TRUE**
deletable	**SD_FALSE**
monitor	**SD_FALSE**
start_arg	null
start_len	0

ms_add_pi returns a pointer to the structure. The user can then use to access the members and change the values, if necessary.

Function Prototype:

```
struct prog_inv *ms_add_pi  (ST_CHAR *piname,
                             ST_INT num_doms,
                             ST_CHAR **doms,
                             ST_UCHAR protection);
```

Parameters:

piname This is the pointer to the name of the program invocation to be created.

num_doms This is the number of elements in the **doms** array.

doms This is an array of pointers to the names of existing named domains to be associated with this program invocation.

protection The value passed in this parameter is placed in the **protection** member of the **PROG_INV** structure corresponding to the program invocation that was added.

Return Value:

PROG_INV * This pointer to the structure of type **PROG_INV** is assigned to hold the information about this program invocation. In case of an error, the pointer is set to null and **mms_op_err** is written with the error code.

ms_change_pi_state

Usage: This support function allows a user to change the state of a program invocation for local reasons. This can occur for several reasons:

1. VMD has been programmed to start a program internally,
2. end of a program has occurred, or
3. an internal stop has occurred from which the program can be restarted at that point.

This function needs to be called with the appropriate arguments when a program changes state internally to the VMD. Then, the MMS-EASE virtual machine is kept up-to-date on the true state of the VMD's program invocations. This function returns either SUCCESS or FAILURE. To properly reference the program invocation, the **`m_vmd_select`** pointer must be set to the correct VMD control structure. See page 2-10 for more information on this structure.

Function Prototype:

```
ST_RET ms_change_pi_state (ST_CHAR *piname,
                           ST_CHAR state);
```

Parameters:

`piname` This is a pointer to the name of the program invocation to be changed.

`state` This is the new state for the specified program invocation. The following values are to be used:

`PI_NON_EXISTENT`
`PI_UNRUNNABLE`
`PI_IDLE`
`PI_RUNNING`
`PI_STOPPED`
`PI_STARTING`
`PI_STOPPING`
`PI_RESUMING`
`PI_RESETTING`

Return Value: `ST_RET` **`SD_SUCCESS`**. No Error.

`<> 0` Error Code.

ms_check_pi_state

Usage: This support function is used by the application to check that the state and other attributes of a program invocation are appropriate for carrying out an operation requested over the network. **mv_xxxx_resp** functions do not check the program invocation state as a criterion for responding positively. So, this function needs to be called before these functions are used, or before the user carries out the requested operation. This function returns SUCCESS if the program invocation state is appropriate; otherwise, an error code is returned. To properly reference the program invocation, the **m_vmd_select** pointer must be set to the correct VMD control structure. See page 2-10 for more information on this structure.

Function Prototype:

```
ST_RET ms_check_pi_state (ST_CHAR *piname,
                          ST_INT op,
                          ST_UCHAR priv);
```

Parameters:

`piname` This is a pointer to the name of the program invocation to be checked for its state.

`op` This is the opcode of the MMS Program Invocation Management service. Normally this is the same as the opcode taken from the **MMSREQ_IND** structure.

`priv` This bitmask is used to detect program invocation protection violations. It must match at least one set bit in the **protection** member of the **PROG_INV** structure. A PROTECTION error will be returned if the **priv** or **protection** members are **0x00**.

Return Value: `ST_RET`

SD_SUCCESS. No Error.

<> 0 Error Code. The following errors are specific to Program Invocations:

PI_NAME	**200**	A Program Invocation Object by that name already exists or a Program Invocation Object should exist but it doesn't.
PI_STATE	**201**	A bad PI state is returned for a given operation.
PI_NOT_REUSABLE	**202**	Cannot reset this Program Invocation.
PI_NOT_DELETABLE	**203**	Cannot delete this Program Invocation.
PI_PROTECTION	**204**	Privilege is invalid to manage this Program Invocation.
INVALID_ID	**205**	Invalid MMS Identifier.

ms_del_all_pis

Usage: This support function deletes an entire list of **PROG_INV** structures from the specified program invocation list. See page 2-379 for more information on this structure.

Function Prototype: `ST_VOID ms_del_all_pis (PROG_INV *pi_list);`

Parameters:

`pi_list` This pointer of structure type **PROG_INV** points to the head of list of program invocations to be deleted.

Return Value: `ST_VOID` (ignored)

ms_del_pi

Usage: This support function deletes an existing entry from the list of **PROG_INV** structures. See page 2-379 for more information on this structure. For each associated **named_domain**, the **npi** member of this structure is decremented. When this function finds that there are no more program invocations that reference the associated domain, the domain state is set to **DOM_READY**. If a **start_arg** is present, and the user has dynamically allocated the space for it, the user should free the pointer before calling this function. The function returns a value of either SUCCESS or FAILURE.

Function Prototype:

```
ST_RET ms_del_pi (ST_CHAR *piname);
```

Parameters:

`piname` This is a pointer to the name of the program invocation to be deleted.

Return Value:

`ST_RET`	**SD_SUCCESS**. No Error.
	`<> 0` Error Code.

ms_find_pi

Usage: This support function is used to locate a program invocation structure within the MMS-EASE database. The function returns a null if the program invocation is not found.

Function Prototype: `struct prog_inv *ms_find_pi (ST_CHAR *piname);`

Parameters:

`piname` This points to the name of the program invocation for which the **`PROG_INV`** structure is to be found. See page 2-379 for more information on this structure.

Return Value:

`PROG_INV *` This pointer to the structure of type **`PROG_INV`** corresponds to the specified **`piname`**. In case of an error, the pointer is returned **`= NULL`** and **`mms_op_err`** is written with the error code.

20. CreateProgramInvocation Service

This service is used to create a specified program invocation at the server. The client specifies a list of domains that are included in this program invocation. Domains can be present in more than one program invocation if the sharable attribute is set.

Primitive Level CreateProgramInvocation Operations

The following section contains information on how to use the paired primitive interface for the CreateProgramInvocation service. It covers available data structures used by the PPI, and the four primitive level functions that together make up the CreateProgramInvocation service. See **Volume 1 — Module 2 — General Sequence of Events — Confirmed Services** for a detailed explanation of the sequence of events that occur within the MMS-EASE environment during confirmed services.

- The CreateProgramInvocation service consists of the paired primitive functions of **`mp_crepi`**, **`u_crepi_ind`**, **`mp_crepi_resp`**, and **`u_mp_crepi_conf`**.

Data Structures

Request/Indication

The operation specific structure described below is used by the Client in issuing a CreateProgramInvocation request (**`mp_crepi`**). It is received by the Server when a CreateProgramInvocation indication (**`u_crepi_ind`**) is received.

```
struct crepi_req_info
  {
  ST_CHAR     piname [MAX_IDENT_LEN+1];
  ST_INT      num_of_dnames;
  ST_BOOLEAN reusable;
  ST_BOOLEAN monitor_pres;
  ST_BOOLEAN monitor;
/*ST_CHAR    *dnames_list [num_of_dnames];                */
  SD_END_STRUCT
  };
typedef struct crepi_req_info CREPI_REQ_INFO;
```

Fields:

`piname` This contains the name of the program invocation being created

`num_of_dnames` This indicates the number of domains to be included in this program invocation.

`reusable` **`SD_TRUE`**. This program invocation can be reset and started from the beginning. (This is not the same as resetting).

`SD_FALSE`. This program invocation cannot be reset and started from the beginning.

`monitor_pres` **`SD_FALSE`**. Do Not include **`monitor`** in the PDU.

`SD_TRUE`. Include **`monitor`** in the PDU.

`monitor` **`SD_TRUE`**. The monitoring of the program invocation should last for the life of the program invocation. This is referred to as permanent monitoring.

`SD_FALSE`. The monitoring of the program invocation should only exist for the life of the association. This is referred to as current monitoring.

`dnames_list`	This is a list of pointers to existing domains to be incorporated as part of this program invocation.

NOTE: FOR REQUEST ONLY, when allocating a data structure of type **`CREPI_REQ_INFO`,** enough memory must be allocated to hold the information for the **`dnames_list`** members of the structure. The following C statement can be used:

```
info = (CREPI_REQ_INFO *) chk_malloc (sizeof (CREPI_REQ_INFO) + num_of_dnames *
              sizeof(ST_CHAR *))
```

Paired Primitive Interface Functions

mp_crepi

Usage: This primitive request function sends a CreateProgramInvocation request PDU to a remote node. It uses the data found in a structure of type **CREPI_REQ_INFO**, pointed to by **info**. This service is used to ask the remote node to assemble a list of domains into a runnable program invocation. Once a program invocation is defined, start, stop, resume, reset, and kill service requests can then be issued for that program invocation. Some devices may come with preprogrammed program invocations that do not need to be created using a Create ProgramInvocation request. Consult the documentation for that particular device for more information.

Function Prototype:

```
MMSREQ_PEND * mp_crepi (ST_INT chan,
                        CREPI_REQ_INFO *info);
```

Parameters:

`chan` This is the channel number over which the CreateProgramInvocation request PDU is to be sent.

`info` This pointer to an operation-specific data structure of type **CREPI_REQ_INFO** contains information specific to the request PDU to be sent.

Return Value:

`MMSREQ_PEND *` This pointer to the request data structure of type **MMSREQ_PEND** is used to send the PDU. In case of an error, the pointer is set to null and **mms_op_err** is written with the error code.

Corresponding User Confirmation Function: **u_mp_crepi_conf**

Operation-Specific Data Structure Used: **CREPI_REQ_INFO**

See page 2-387 for a detailed description of this structure.

u_crepi_ind

Usage: This user indication function is called when a CreateProgramInvocation is received by a Server node. The contents and operation of this function are user-defined and depend on the application requirements. To conform to the MMS protocol requirements, the application must do one of the following some time after the indication is received:

1) send a positive response using:

   a) the primitive response function (**mp_crepi_resp**) after the specified program invocation has been successfully created,

   b) the virtual machine response function (**mv_crepi_resp**) to have the virtual machine take the requested action and send the response, or

2) call the appropriate primitive negative (error) response function (**mp_err_resp**). See **Volume 3 — Module 11 — MMS-EASE Error Handling** for more information on this function.

See **Volume 1 — Module 1 — User Indication Function Class** for additional information on the recommended handling of indications.

Function Prototype:

```
ST_VOID u_crepi_ind  (MMSREQ_IND *ind);
```

Parameters:

`ind` This pointer to the **MMSREQ_IND** structure is described in **Volume 1 — Module 2 — Request and Indication Control Data Structures**. This pointer must be used to send the response to this indication. The primary item of interest in this structure is **ind->req_info_ptr**. This is a pointer to the operation-specific data structure for a CreateProgramInvocation indication (**CREPI_REQ_INFO**). This pointer will always be valid when **u_crepi_ind** is called.

Return Value: `ST_VOID` (ignored)

Operation-Specific Data Structure Used: **CREPI_REQ_INFO**

See page 2-387 for a detailed description of this structure.

mp_crepi_resp

Usage: This primitive response function sends a CreateProgramInvocation positive response PDU. This function should be called after the **u_crepi_ind** function is called (a CreateProgramInvocation indication was received) and after the specified program invocation has been successfully created.

Function Prototype: `ST_RET mp_crepi_resp (MMSREQ_IND *ind);`

Parameters:

`ind` This pointer to the indication control structure of type **MMSREQ_IND** is passed to the **u_crepi_ind** function when it was called.

Return Value: `ST_RET` **SD_SUCCESS**. No Error.

`<> 0` Error Code.

Corresponding User Indication Function: **u_crepi_ind**

Operation-Specific Data Structure Used: NONE

u_mp_crepi_conf

Usage: This primitive user confirmation function is called when a confirm to a CreateProgramInvocation request (**mp_crepi**) is received. **resp_err** contains a value indicating whether an error occurred. **resp_info_ptr** may contain data that the remote node sent back in the confirmation or it may contain error information if an error occurred. See **Volume 1 — Module 1 — Primitive User Confirmation Function Class** for more information on the recommended handling of confirms.

Note that the function called for this event may be changed if desired; however, **u_mp_crepi_conf** is the standard default function.

Function Prototype:

```
ST_VOID u_mp_crepi_conf (MMSREQ_PEND *req);
```

Parameters:

`req` This pointer to the **MMSREQ_PEND** structure is returned from the original **mp_crepi** request function. See **Volume 1 — Module 1 — Request and Indication Control Data Structures** for more information on this structure.

For a positive response (**req->resp_err = CNF_RESP_OK**), the request control structure member **resp_info_pres** indicates that there is no confirm data for this function.

For a negative response, or other errors, see **Volume 3 — Module 11 — MMS-EASE Error Handling** for more information.

Return Value: `ST_VOID` (ignored)

Operation-Specific Data Structure Used: NONE

Virtual Machine CreateProgramInvocation Operations

The following section contains information on how to use the virtual machine interface for the Create Program Invocation service. It covers the virtual machine response function used.

- The CreateProgramInvocation consists of the virtual machine function of **mv_crepi_resp**.

There are no virtual machine level data structures.

Virtual Machine Interface Functions

mv_crepi_resp

Usage: This virtual machine response function allows the user to send a CreateProgramInvocation response without having to create explicitly an input structure of type **CREPI_RESP_INFO**. Before calling this function, the user must call the support function, **ms_check_pi_state**. This makes sure that the program invocation is in the correct state, and that the application can actually perform this service. See page 2-311 for more information on this support function.

Function Prototype: `ST_RET mv_crepi_resp (MMSREQ_IND *ind, ST_RET err);`

Parameters:

`ind` This pointer to the indication control data structure of type **MMSREQ_IND** is passed to the **u_crepi_ind** function when it was called.

`err` This is the value normally derived by calling **ms_check_pi_state**. When this argument is not equal to zero (<>0), this value indicates the reason for sending a negative response. The following errors are specific to Program Invocations:

PI_NAME	**200**	A Program Invocation Object by that name already exists or a Program Invocation Object should exist but it doesn't.
PI_STATE	**201**	A bad PI state is returned for a given operation.
PI_NOT_REUSABLE	**202**	Cannot reset this Program Invocation.
PI_NOT_DELETABLE	**203**	Cannot delete this Program Invocation.
PI_PROTECTION	**204**	Privilege is invalid to manage this Program Invocation.
INVALID_ID	**205**	Invalid MMS Identifier.

Return Value: `ST_RET` **SD_SUCCESS**. No Error.

<> 0 Error Code.

21. DeleteProgramInvocation Service

This service is used to delete a specified existing program invocation at a MMS Server.

Primitive Level DeleteProgramInvocation Operations

The following section contains information on how to use the paired primitive interface for the DeleteProgramInvocation service. It covers available data structures used by the PPI, and the four primitive level functions that together make up the DeleteProgramInvocation service. See **Volume 1 — Module 2 — General Sequence of Events — Confirmed Services** for a detailed explanation of the sequence of events that occur within the MMS-EASE environment during confirmed services.

- The DeleteProgramInvocation service consists of the paired primitive functions of **`mp_delpi`**, **`u_delpi_ind`**, **`mp_delpi_resp`**, and **`u_mp_delpi_conf`**.

Data Structures

Request/Indication

The operation specific structure described below is used by the Client in issuing a DeleteProgramInvocation request (**`mp_delpi`**). It is received by the Server when a DeleteProgramInvocation indication (**`u_delpi_ind`**) is received.

```
struct delpi_req_info
  {
  ST_CHAR piname [MAX_IDENT_LEN+1];
  };
typedef struct delpi_req_info DELPI_REQ_INFO;
```

Fields:

`piname` This contains the name of the program invocation to be deleted.

Paired Primitive Interface Functions

mp_delpi

Usage: This primitive request function sends a DeleteProgramInvocation request PDU to a remote node. It uses the data found in a structure of type **DELPI_REQ_INFO**, pointed to by **info**. This service is used to request that the remote node delete the specified program invocation.

Function Prototype:

```
MMSREQ_PEND *mp_delpi (ST_INT chan,
                       DELPI_REQ_INFO *info);
```

Parameters:

chan	This is the channel number over which the DeleteProgramInvocation request is to be sent.
info	This pointer to an Operation-Specific data structure of type **DELPI_REQ_INFO** contains information specific to the request PDU to be sent.

Return Value:

MMSREQ_PEND *	This pointer to the request control data structure of type **MMSREQ_PEND** is used to send the PDU. In case of an error, the pointer is set to null and **mms_op_err** is written with the error code.

Corresponding User Confirmation Function: **u_mp_delpi_conf**

Operation-Specific Data Structure Used: **DELPI_REQ_INFO**

See page 2-395 for a detailed description of this structure.

u_delpi_ind

Usage: This user indication function is called when a DeleteProgramInvocation indication is received by a Server node. The contents and operation of this function are user-defined and depend on the application requirements. To conform to the MMS protocol requirements, the application must do one of the following some time after the indication is received:

1) send a positive response using:

   a) the primitive response function (**`mp_delpi_resp`**) after the specified program invocation has been successfully deleted,

   b) the virtual machine response function (**`mv_delpi_resp`**) to have the virtual machine take the requested action and send the response, or

2) call the appropriate primitive negative (error) response function (**`mp_err_resp`**). See **Volume 3 — Module 11 — MMS-EASE Error Handling** for more information on this function.

See **Volume 1 — Module 1 — User Indication Function Class** for additional information on the recommended handling of indications.

Function Prototype: `ST_VOID u_delpi_ind (MMSREQ_IND *ind);`

Parameters:

`ind` This pointer to the **`MMSREQ_IND`** structure is described in **Volume 1 — Module 2 — Request and Indication Control Data Structures**. This pointer must be used to send the response to this indication. The primary item of interest in this structure is **`ind->req_info_ptr`**. This is a pointer to the operation-specific data structure for a DeleteProgramInvocation indication (**`DELPI_REQ_INFO`**). This pointer will always be valid when **`u_delpi_ind`** is called.

Return Value: `ST_VOID` (ignored)

Operation-Specific Data Structure Used: **`DELPI_REQ_INFO`**

See page 2-395 for a detailed description of this structure.

mp_delpi_resp

Usage: This primitive response function is used for sending a DeleteProgramInvocation positive response PDU. This function should be called after the **u_delpi_ind** function is called (a DeleteProgramInvocation indication was received), and after the specified program invocation (which should already exist) has been successfully deleted. Program invocations should only be deleted if they are in the **PI_IDLE**, **PI_STOPPED**, or **PI_UNRUNNABLE** state, and have the MMS deletable attribute set. Any event conditions, event actions, and event enrollments associated with a program invocation with the monitored attribute set also should be deleted. You should change the state of any domains that are only used by this program invocation (not sharable) from the **DOM_IN_USE** state to the **DOM_READY** state.

Function Prototype: `ST_RET mp_delpi_resp (MMSREQ_IND *ind);`

Parameters:

`ind` This pointer to the indication control data structure of type **MMSREQ_IND** is passed to the **u_delpi_ind** function when it was called.

Return Value: `ST_RET` **SD_SUCCESS**. No Error.

`<> 0` Error Code.

Corresponding User Indication Function: **u_delpi_ind**

Operation-Specific Data Structure Used: NONE

u_mp_delpi_conf

Usage: This primitive user confirmation function is called when a confirm to a DeleteProgramInvocation request (**mp_delpi**) is received. **resp_err** contains a value indicating whether an error occurred. **resp_info_ptr** may contain data that the remote node sent back in the confirmation or it may contain error information if an error occurred. See **Volume 1 — Module 1 — Primitive User Confirmation Function Class** for more information on the recommended handling of confirms.

Note that the function called for this event may be changed if desired; however, **u_mp_delpi_conf** is the standard default function.

Function Prototype:

```
ST_VOID u_mp_delpi_conf (MMSREQ_PEND *req);
```

Parameters:

`req` This pointer to the **MMSREQ_PEND** structure is returned from the original **mp_delpi** request function. See **Volume 1 — Module 2 — Request and Indication Control Data Structures** for more information on this structure.

For a positive response (**req->resp_err = CNF_RESP_OK**), the request control structure member **resp_info_pres** indicates that there is no confirm data for this function.

For a negative response, or other errors, see **Volume 3 — Module 11 — MMS-EASE Error Handling** for more information.

Return Value: `ST_VOID` (ignored)

Operation-Specific Data Structure Used: NONE

Virtual Machine DeleteProgramInvocation Operations

The following section contains information on how to use the virtual machine interface for the DeleteProgramInvocation service. It covers the virtual machine response function used. There are no virtual machine level data structures.

- The DeleteProgramInvocation consists of the virtual machine function of **mv_delpi_resp**.

Virtual Machine Interface Functions

mv_delpi_resp

Usage: This virtual machine response function allows the user to send a DeleteProgramInvocation response without having to create explicitly an input structure of type **DELPI_RESP_INFO**. Before calling this function, the user must call the support function, **ms_check_pi_state**. This makes sure that the program invocation is in the correct state, and that the application can actually perform this service BEFORE calling this function. See page 2-311 for more information on this support function. MMS-EASE assigns all channels to reference the default **VMD_CTRL** structure when **strt_MMS()** is called. If more than one **VMD_CTRL** structure is being used, BE SURE TO SET **mms_chan_info.objs.vmd** TO POINT TO THE PROPER VMD STRUCTURE BEFORE CALLING THIS FUNCTION. See **Volume 1 — Module 2 — Channel Information Structure** and **Module 4 — VMD Control** starting on page 2-10 for more information.

Function Prototype:

```
ST_RET mv_delpi_resp (MMSREQ_IND *indptr,
                      ST_RET err_reason,
                      ST_BOOLEAN soft);
```

Parameters:

`ind` This pointer to the indication control data structure of type **MMSREQ_IND** is passed to the **u_delpi_ind** function when it was called.

`err` This is the value normally derived by calling **ms_check_pi_state**. When this argument is not equal to zero (`<> 0`), this value indicates the reason for sending a negative response. The following errors are specific to Program Invocations:

PI_NAME	**200**	A Program Invocation Object by that name already exists or a Program Invocation Object should exist but it doesn't.
PI_STATE	**201**	A bad PI state is returned for a given operation.
PI_NOT_REUSABLE	**202**	Cannot reset this Program Invocation.
PI_NOT_DELETABLE	**203**	Cannot delete this Program Invocation.
PI_PROTECTION	**204**	Privilege is invalid to manage this Program Invocation.
INVALID_ID	**205**	Invalid MMS Identifier.

`soft` **SD_TRUE**. A soft delete is performed. In other words, the **PROG_INV** structure is not actually deleted; only the state attribute is set to **NON_EXISTENT**.

SD_FALSE . The **PROG_INV** structure is deleted. All buffers are returned to the heap.

Return Value: `ST_RET` **SD_SUCCESS**. No Error.

`<> 0` Error Code.

22. Start Service

This service is used to allow the client to change the state of a specified program invocation to the **PI_RUNNING** state at the server. This program invocation must be in the **PI_IDLE** state.

Primitive Level Start Operations

The following section contains information on how to use the paired primitive interface for the Start service. It covers available data structures used by the PPI, and the four primitive level functions that together make up the Start service. See **Volume 1 — Module 2 — General Sequence of Events — Confirmed Services** for a detailed explanation of the sequence of events that occur within the MMS-EASE environment during confirmed services.

- The Start service consists of the paired primitive functions of **mp_start**, **u_start_ind**, **mp_start_resp**, and **u_mp_start_conf**.

Data Structures

Request/Indication

The operation specific structure described below is used by the Client in issuing a Start request (**mp_start**). It is received by the Server when a Start indication (**u_start_ind**) is received.

```
struct start_req_info
  {
  ST_CHAR     piname [MAX_IDENT_LEN+1];
  ST_INT16    start_arg_type;
  ST_BOOLEAN start_arg_pres;
  ST_INT     start_arg_len;
  ST_UCHAR    *start_arg;
  SD_END_STRUCT
  };
typedef struct start_req_info START_REQ_INFO;
```

Fields:

`piname` This contains the name of the program invocation to use.

`start_arg_type` This indicates the type of argument used to pass data to the program invocation:

SIMPLE_STRING. This is a null-terminated character string.

ENCODED_STRING. This is a value encoded by some other external source - ASN.1 EXTERNAL data element. See page 2-271 for more information on how to encode and decode ASN.1 EXTERNAL data elements.

`start_arg_pres` **SD_FALSE**. Include **start_arg** in the PDU.

SD_TRUE. Do Not include **start_arg** in the PDU.

`start_arg_len` This is the length, in bytes, of the data present in **start_arg**. Used only if **start_arg_type = SD_TRUE**.

`start_arg` This pointer to the start argument is used to pass data to the program invocation.

Paired Primitive Interface Functions

mp_start

Usage: This primitive request function sends a Start request PDU using the data from a structure of type **START_REQ_INFO**, pointed to by **info**. The Start service is used to change the state of a program invocation at the remote node from **PI_IDLE** to **PI_RUNNING**.

Function Prototype:
```
MMSREQ_PEND *mp_start (ST_INT chan,
                       START_REQ_INFO *info);
```

Parameters:

chan	This integer contains the channel number over which the Start PDU will be sent.
info	This pointer to an operation-specific data structure of type **START_REQ_INFO** contains information specific to the Start PDU to be sent.

Return Value:

MMSREQ_PEND *	This pointer to the request control data structure of type **MMSREQ_PEND** is used to send the Start PDU. In case of an error, the pointer is set to null and **mms_op_err** is written with the error code.

Corresponding User Confirmation Function: **u_mp_start_conf**

Operation-Specific Data Structure Used: **START_REQ_INFO**

See page 2-401 for a detailed description of this structure.

u_start_ind

Usage: This user indication function is called when a Start indication is received by a Server node. The contents and operation of this function are user-defined and depend on the application requirements. To conform to the MMS protocol requirements, the application must do one of the following some time after the indication is received:

1) send a positive response using either:

   a) the primitive response function (**mp_start_resp**) after the specified program invocation has been successfully changed from the **PI_IDLE** state to the **PI_RUNNING** state,

   b) the virtual machine response function (**mv_start_resp**) to have the virtual machine take the requested action and send the response, or

2) call the appropriate primitive negative (error) response function (**mp_err_resp**). See **Volume 3 — Module 11 — MMS-EASE Error Handling** for more information on this function.

See **Volume 1 — Module 1 — User Indication Function Class** for additional information on the recommended handling of indications.

Function Prototype:

```
ST_VOID u_start_ind  (MMSREQ_IND *ind);
```

Parameters:

`ind` This pointer to the **MMSREQ_IND** structure is described in **Volume 1 — Module 2 — Request and Indication Control Data Structures**. This pointer must be used to send the response to this indication. The primary item of interest in this structure is **ind->req_info_ptr**. This is a pointer to the operation-specific data structure for a Start indication (**START_REQ_INFO**). This pointer will always be valid when **u_start_ind** is called.

Return Value: `ST_VOID` (ignored)

Operation-Specific Data Structure Used: **START_REQ_INFO**

See page 2-401 for a detailed description of this structure.

mp_start_resp

Usage: This primitive response function sends a Start response PDU. This function should be called after the **u_start_ind** function is called (a Start indication was received), and after the specified program invocation has been successfully changed from the **PI_IDLE** state to the **PI_RUNNING** state.

Function Prototype: `ST_RET mp_start_resp (MMSREQ_IND *ind);`

Parameters:

`ind` This pointer to the indication control structure of type **MMSREQ_IND** is passed to the **u_start_ind** function when it was called.

Return Value: `ST_RET` **SD_SUCCESS**. No Error.

`<> 0` Error Code.

Corresponding User Indication Function: **u_start_ind**

Operation-Specific Data Structure Used: NONE

u_mp_start_conf

Usage: This primitive user confirmation function is called when a confirm to a Start request (**mp_start**) is received. **resp_err** contains a value indicating whether an error occurred. **resp_info_ptr** may contain data that the remote node sent back in the confirmation or it may contain error information if an error occurred. See **Volume 1— Module 1 — Primitive User Confirmation Function Class** for more information on the recommended handling of confirms.

Note that the function called for this event may be changed if desired; however, **u_mp_start_conf** is the standard default function.

Function Prototype:

```
ST_VOID u_mp_start_conf (MMSREQ_PEND *req);
```

Parameters:

`req` This pointer to the **MMSREQ_PEND** structure is returned from the original **mp_start** request function. See **Volume 1 — Module 2 — Request and Indication Control Data Structures** for more information on this structure.

For a positive response (**req->resp_err = CNF_RESP_OK**), the request control structure member **resp_info_pres** indicates that there is no confirm data for this function.

For a negative response, or other errors, see **Volume 3 — Module 11 — MMS-EASE Error Handling** for more information.

Return Value: `ST_VOID` (ignored)

Operation-Specific Data Structure Used: NONE

Virtual Machine Start Operations

The following section contains information on how to use the virtual machine interface for the Start service. It covers the virtual machine response function used.

- The Start service consists of the virtual machine function of **mv_start_resp**.

There is no virtual machine level data structures.

Virtual Machine Interface Functions

mv_start_resp

Usage: This virtual machine response function allows the user to respond to a Start response without having to create explicitly an input structure of type **START_RESP_INFO**. Before calling this function, the user must use the support function, **ms_check_pi_state**. This makes sure that the program invocation is in the correct state, and that the application program can actually perform this service. See page 2-311 for more information on this support function. This service returns a zero (0) if it is a SUCCESS and also updates the **PROG_INV** structure. If an error occurs, **mms_op_err** code is used if MMS-EASE could not send the response. See **Volume 1 — Appendix A — Error Codes** for more information on this code's possible values. MMS-EASE assigns all channels to reference the default **VMD_CTRL** structure when **strt_MMS()** is called. If more than one **VMD_CTRL** structure is being used, BE SURE TO SET **mms_chan_info.objs.vmd** TO POINT TO THE PROPER VMD STRUCTURE BEFORE CALLING THIS FUNCTION. See **Volume 1 — Module 2 — Channel Information Structure** and **Module 4 — VMD Control** starting on page 2-10 for more information.

Function Prototype: `ST_RET mv_start_resp (MMSREQ_IND *ind, ST_RET err);`

Parameters:

`ind` This pointer to the indication control data structure of type **MMSREQ_IND** is passed to the **u_start_ind** function when it was called.

`err` This is the value normally derived by calling the **ms_check_pi_state** function. When this argument is NOT equal to zero (`<> 0`), this value indicates the reason for sending a negative response. The following errors are specific to Program Invocations:

PI_NAME	**200**	A Program Invocation Object by that name already exists or a Program Invocation Object should exist but it doesn't.
PI_STATE	**201**	A bad PI state is returned for a given operation.
PI_NOT_REUSABLE	**202**	Cannot reset this Program Invocation.
PI_NOT_DELETABLE	**203**	Cannot delete this Program Invocation.
PI_PROTECTION	**204**	Privilege is invalid to manage this Program Invocation.
INVALID_ID	**205**	Invalid MMS Identifier.

Return Value: `ST_RET` **SD_SUCCESS**. No Error.

`<> 0` Error Code.

23. Stop Service

This service is used to cause a specified program invocation at the server to transition out of the **PI_RUNNING** state and into the **PI_STOPPED** state.

Primitive Level Stop Operations

The following section contains information on how to use the paired primitive interface for the Stop service. It covers available data structures used by the PPI, and the four primitive level functions that together make up the Stop service. See **Volume 1 — Module 2 — General Sequence of Events — Confirmed Services** for a detailed explanation of the sequence of events that occur within the MMS-EASE environment during confirmed services.

- The Stop service consists of the paired primitive functions of **mp_stop**, **u_stop_ind**, **mp_stop_resp**, and **u_mp_stop_conf**.

Data Structures

Request/Indication

The operation specific structure described below is used by the Client in issuing a Stop request (**mp_stop**). It is received by the Server when a Stop indication (**u_stop_ind**) is received.

```
struct stop_req_info
  {
  ST_CHAR piname [MAX_IDENT_LEN+1];
  };
typedef struct stop_req_info STOP_REQ_INFO;
```

Fields:

`piname` This contains the name of the program invocation to be stopped.

Paired Primitive Interface Functions

mp_stop

Usage: This primitive request function sends a Stop request PDU using the data from a structure of type **STOP_REQ_INFO**, pointed to by **info**. The Stop service is used to request that a named program invocation at the remote node leave the **PI_RUNNING** state and go into the **PI_STOPPED** state.

Function Prototype: `MMSREQ_PEND *mp_stop (ST_INT chan, STOP_REQ_INFO *info);`

Parameters:

`chan`	This integer contains the channel number over which the stop PDU is to be sent.
`info`	This pointer to an operation-specific data structure of type **STOP_REQ_INFO** contains data specific to the Stop PDU to be sent.

Return Value:

`MMSREQ_PEND *`	This pointer to the request control data structure of type **MMSREQ_PEND** is used to send the stop PDU. In case of an error, the pointer is set to null and **mms_op_err** is written with the error code.

Corresponding User Confirmation Function: **u_mp_stop_conf**

Operation-Specific Data Structure Used: **STOP_REQ_INFO**

See page 2-407 for a detailed description of this structure.

u_stop_ind

Usage: This user indication function is called when a Stop indication is received by a SERVER node. The contents and operation of this function are user-defined and depend on the application requirements. To conform to the MMS protocol requirements, the application must do one of the following some time after the indication is received:

1) send a positive response using either:

   a) the primitive response function (**mp_stop_resp**) after the specified program invocation has been successfully changed from the **PI_RUNNING** state to the **PI_STOPPED** state,

   b) the virtual machine response function (**mv_stop_resp**) to have the virtual machine take the requested action and send the response, or

2) call the appropriate primitive negative (error) response function (**mp_err_resp**). See **Volume 3 — Module 11 — MMS-EASE Error Handling** for more information on this function.

See **Volume 1 — Module 1 — User Indication Function Class** for additional information on the recommended handling of indications.

Function Prototype: `ST_VOID u_stop_ind (MMSREQ_IND *ind);`

Parameters:

`ind` This pointer to the **MMSREQ_IND** structure is described in **Volume 1 — Module 2 — Request and Indication Control Data Structures**. This pointer must be used to send the response to this indication. The primary item of interest in this structure is **ind->req_info_ptr**. This is a pointer to the operation-specific data structure for a Stop indication (**STOP_REQ_INFO**). This pointer will always be valid when **u_stop_ind** is called.

Return Value: `ST_VOID` (ignored)

Operation-Specific Data Structure Used: **STOP_REQ_INFO**

See page 2-407 for a detailed description of this structure.

mp_stop_resp

Usage: This primitive response function sends a Stop positive response PDU. This function should be called after the **u_stop_ind** function is called (a Stop indication was received), and after the specified program invocation has been successfully changed from the **PI_RUNNING** state to the **PI_STOPPED** state.

Function Prototype: `ST_RET mp_stop_resp (MMSREQ_IND *ind);`

Parameters:

`ind` This pointer to the indication control structure of type **MMSREQ_IND** is passed to the **u_stop_ind** function when it was called.

Return Value: `ST_RET` **SD_SUCCESS**. No Error.

`<> 0` Error Code.

Corresponding User Indication Function: **u_stop_ind**

Operation-Specific Data Structure Used: NONE

u_mp_stop_conf

Usage: This primitive user confirmation function is called when a confirm to a Stop request (**mp_stop**) is received. **resp_err** contains a value indicating whether an error occurred. **resp_info_ptr** may contain data that the remote node sent back in the confirmation, or it may contain error information if an error occurred. See **Volume 1 — Module 1 — Primitive User Confirmation Function Class** for more information on the recommended handling of confirms.

Note that the function called for this event may be changed if desired; however, **u_mp_stop_conf** is the standard default function.

Function Prototype:

```
ST_VOID u_mp_stop_conf (MMSREQ_PEND *req);
```

Parameters:

`req` This pointer to the **MMSREQ_PEND** structure is returned from the original **mp_stop** request function. See **Volume 1 — Module 2 — Request and Indication Control Data Structures** for more information on this structure.

For a positive response (**req->resp_err = CNF_RESP_OK**), the request control structure member **resp_info_pres** indicates that there is no confirm data for this function.

For a negative response, or other errors, see **Volume 3 — Module 11 — MMS-EASE Error Handling** for more information.

Return Value: `ST_VOID` (ignored)

Operation-Specific Data Structure Used: NONE

Virtual Machine Stop Operations

The following section contains information on how to use the virtual machine interface for the Stop service. It covers the virtual machine response function used.

- The Stop service consists of the virtual machine function of **mv_stop_resp**.

There are no virtual machine level data structures.

Virtual Machine Interface Functions

mv_stop_resp

Usage: This virtual machine function allows the user to send a Stop response without having to create explicitly an input structure of type **STOP_RESP_INFO**. Before calling this function, the user must call the support function, **ms_check_pi_state**. This makes sure that the program invocation is in the correct state, and that the application program can actually perform this service. See page 2-383 for more information on this support function. This service returns a zero (0) if it is a SUCCESS and also updates the **PROG_INV** structure. If an error occurs, **mms_op_err** code is used if MMS-EASE could not send the response. See **Volume 1 — Appendix A** for more information on this code's possible values. MMS-EASE assigns all channels to reference the default **VMD_CTRL** structure when **strt_MMS()** is called. If more than one **VMD_CTRL** structure is being used, BE SURE TO SET **mms_chan_info.objs.vmd** TO POINT TO THE PROPER VMD STRUCTURE BEFORE CALLING THIS FUNCTION. See **Volume 1 — Module 2 — Channel Information Structure** and **Module 4 — VMD Control** starting on page 2-10 for more information.

Function Prototype: `ST_RET mv_stop_resp (MMSREQ_IND *ind, ST_RET err);`

Parameters:

`ind` This pointer to the indication control data structure of type **MMSREQ_IND** is passed to the **u_stop_ind** function when it was called.

`err` This is the value normally derived by calling the **ms_check_pi_state** function. When this argument is NOT equal to zero (`<> 0`), this value indicates the reason for sending a negative response. The following errors are specific to Program Invocations:

PI_NAME	**200**	A Program Invocation Object by that name already exists or a Program Invocation Object should exist but it doesn't.
PI_STATE	**201**	A bad PI state is returned for a given operation.
PI_NOT_REUSABLE	**202**	Cannot reset this Program Invocation.
PI_NOT_DELETABLE	**203**	Cannot delete this Program Invocation.
PI_PROTECTION	**204**	Privilege is invalid to manage this Program Invocation.
INVALID_ID	**205**	Invalid MMS Identifier.

Return Value: `ST_RET` **SD_SUCCESS**. No Error.

`<> 0` Error Code.

24. Resume Service

This service is used to cause the state of a specified program invocation at the server to be changed to the **PI_RUNNING** state. This program invocation must be in the **PI_STOPPED** state in order for this service to be valid.

Primitive Level Resume Operations

The following section contains information on how to use the paired primitive interface for the Resume service. It covers available data structures used by the PPI, and the four primitive level functions that together make up the Resume service. See **Volume 1 — Module 2 — General Sequence of Events — Confirmed Services** for a detailed explanation of the sequence of events that occur within the MMS-EASE environment during confirmed services.

- The Resume service consists of the paired primitive functions of **mp_resume**, **u_resume_ind**, **mp_resume_resp**, and **u_mp_resume_conf**.

Data Structures

Request/Indication

The operation specific structure described below is used by the Client in issuing a Resume request (**mp_resume**). It is received by the Server when a Resume indication (**u_resume_ind**) is received.

```
struct resume_req_info
  {
  ST_CHAR     piname [MAX_IDENT_LEN+1];
  ST_INT16    resume_arg_type;
  ST_BOOLEAN resume_arg_pres;
  ST_INT      resume_arg_len;
  ST_UCHAR    *resume_arg;
  };
typedef struct resume_req_info RESUME_REQ_INFO;
```

Fields:

`piname` This contains the name of the program invocation to be resumed.

`resume_arg_type` This indicates the type of argument used to pass data to the program invocation:

SIMPLE_STRING. This is a null-terminated character string.

ENCODED_STRING. This is a value encoded by some other external source - ASN.1 EXTERNAL data element. See page 2-271 for more information on how to encode and decode ASN.1 EXTERNAL data elements.

`resume_arg_pres` **SD_TRUE**. Include **resume_arg** in the PDU.

SD_FALSE. Do Not include **resume_arg** in the PDU.

`resume_arg_len` This length, in bytes, of the data present in **resume_arg** is used only if **resume_arg_type = SD_TRUE**.

`resume_arg` This pointer to the resume argument is used to pass data to the program invocation.

Paired Primitive Interface Functions

mp_resume

Usage: This primitive request function sends a Resume request PDU using the data from a structure of type **RESUME_REQ_INFO**, pointed to by **info**. The Resume service is used to cause the specified program invocation at the remote node to leave the **PI_STOPPED** state and to enter the **PI_RUNNING** state. The specified program invocation at the remote node MUST be in the **PI_STOPPED** state.

Function Prototype:

```
MMSREQ_PEND *mp_resume (ST_INT chan,
                        RESUME_REQ_INFO *info);
```

Parameters:

chan	This is the channel number over which the Resume PDU is to be sent.
info	This pointer to an Operation-Specific data structure of type **RESUME_REQ_INFO** contains information specific to the Resume PDU to be sent.

Return Value:

MMSREQ_PEND *	This pointer to the request control data structure of type **MMSREQ_PEND** is used to send the Resume PDU. In case of an error, the pointer is set to null and **mms_op_err** is written with the error code.

Corresponding User Confirmation Function: **u_mp_resume_conf**

Operation-Specific Data Structure Used: **RESUME_REQ_INFO**

See page 2-413 for a detailed description of this structure.

u_resume_ind

Usage: This user indication function is called when a Resume indication is received by a Server node. The contents and operation of this function are user-defined and depend on the application requirements. To conform to the MMS protocol requirements, the application must do one of the following some time after the indication is received:

1) send a positive response using either:

   a) the primitive response function (**mp_resume_resp**) after the state of the specified program invocation has been successfully changed from **PI_STOPPED** to **PI_RUNNING**.

   b) the virtual machine response function (**mv_resume_resp**) to have the virtual machine take the requested action and send the response, or

2) call the appropriate primitive negative (error) response function (**mp_err_resp**). See **Volume 3 — Module 11 — MMS-EASE Error Handling** for more information on this function.

See **Volume 1 — Module 1 — User Indication Function Class** for additional information on the recommended handling of indications.

Function Prototype: `ST_VOID u_resume_ind (MMSREQ_IND *ind);`

Parameters:

`ind` This pointer to the **MMSREQ_IND** structure is described in **Volume 1 — Module 2 — Request and Indication Control Data Structures**. This pointer must be used to send the response to this indication. The primary item of interest in this structure is **ind->req_info_ptr**. This is a pointer to the operation-specific data structure for a Resume indication (**RESUME_REQ_INFO**). This pointer will always be valid when **u_resume_ind** is called.

Return Value: `ST_VOID` (ignored)

Operation-Specific Data Structure Used: RESUME_REQ_INFO

See page 2-413 for a detailed description of this structure.

mp_resume_resp

Usage: This primitive response function sends a Resume positive response PDU. It should be called after the **u_resume_ind** function is called (a Resume indication was received), and after the state of the specified program invocation has been successfully changed from **PI_STOPPED** to **PI_RUNNING**.

Function Prototype: `ST_RET mp_resume_resp (MMSREQ_IND *ind);`

Parameters:

`ind` This pointer to the indication control structure of type **MMSREQ_IND** is passed to the **u_resume_ind** function when it was called.

Return Value: `ST_RET` **SD_SUCCESS**. No Error.

`<> 0` Error Code.

Corresponding User Indication Function: **u_resume_ind**

Operation-Specific Data Structure Used: NONE

u_mp_resume_conf

Usage: This primitive user confirmation function is called when a confirm to a Resume request (**mp_resume**) is received. **resp_err** contains a value indicating whether an error occurred. **resp_info_ptr** may contain data that the remote node sent back in the confirmation, or it may contain error information if an error occurred. See **Volume 1 — Module 1 — Primitive User Confirmation Function Class** for more information on the recommended handling of confirms.

Note that the function called for this event may be changed if desired; however, **u_mp_resume_conf** is the standard default function.

Function Prototype:

```
ST_VOID u_mp_resume_conf (MMSREQ_PEND *req);
```

Parameters:

`req` This pointer to the **MMSREQ_PEND** structure is returned from the original **mp_resume** request function. See **Volume 1 — Module 2 — Request and Indication Control Data Structures** for more information on this structure.

For a positive response (**req->resp_err = CNF_RESP_OK**), the request control structure member **resp_info_pres** indicates that there is no confirm data for this function.

For a negative response, or other errors, see **Volume 3 — Module 11 — MMS-EASE Error Handling** for more information.

Return Value: `ST_VOID` (ignored)

Operation-Specific Data Structure Used: NONE

Virtual Machine Resume Operations

The following section contains information on how to use the virtual machine interface for the Resume service.

- The Resume service consists of the virtual machine function of **mv_resume_resp**.

There are no virtual machine level data structures.

Virtual Machine Interface Functions

mv_resume_resp

Usage: This virtual machine function allows the user to respond to a Resume response without having to create explicitly an input structure of type **RESUME_RESP_INFO**. Before calling this function, the user must call the support function, **ms_check_pi_state**. It makes sure that the program invocation is in the correct state, and that the application program can actually perform this service. See page 2-311 for more information on this support function. This service returns a zero (0) if it is a SUCCESS and also updates the **PROG_INV** structure. If an error occurs, **mms_op_err** code is used if MMS-EASE could not send the response. See **Volume 1 — Appendix A** for more information on this code's possible values. MMS-EASE assigns all channels to reference the default **VMD_CTRL** structure when **strt_MMS()** is called. If more than one **VMD_CTRL** structure is being used, BE SURE TO SET **mms_chan_info.objs.vmd** TO POINT TO THE PROPER VMD STRUCTURE BEFORE CALLING THIS FUNCTION. See **Volume 1 — Module 2 — Channel Information Structure** and **Module 4 — VMD Control** starting on page 2-10 for more information.

Function Prototype: `ST_RET mv_resume_resp (MMSREQ_IND *ind, ST_RET err);`

Parameters:

`ind` This pointer to the indication control data structure of type **MMSREQ_IND** is passed to the **u_resume_ind** function when it was called.

`err` This is the value normally derived by calling the **ms_check_pi_state** function. When this argument is NOT equal to zero (<> 0), then this value indicates the reason for sending a negative response. The following errors are specific to Program Invocations:

PI_NAME	**200**	A Program Invocation Object by that name already exists or a Program Invocation Object should exist but it doesn't.
PI_STATE	**201**	A bad PI state is returned for a given operation.
PI_NOT_REUSABLE	**202**	Cannot reset this Program Invocation.
PI_NOT_DELETABLE	**203**	Cannot delete this Program Invocation.
PI_PROTECTION	**204**	Privilege is invalid to manage this Program Invocation.
INVALID_ID	**205**	Invalid MMS Identifier.

Return Value: `ST_RET` **SD_SUCCESS**. No Error.

`<> 0` Error Code.

25. Reset Service

This service is used to cause the state of a specified program invocation at the Server to transition from the **PI_STOPPED** state to the **PI_IDLE** state.

Primitive Level Reset Operations

The following section contains information on how to use the paired primitive interface for the Reset service. It covers available data structures used by the PPI, and the four primitive level functions that together make up the Reset service. See **Volume 1 — Module 2 — General Sequence of Events — Confirmed Services** for a detailed explanation of the sequence of events that occur within the MMS-EASE environment during confirmed services.

- The Reset service consists of the paired primitive functions of **mp_reset**, **u_reset_ind, mp_reset_resp,** and **u_mp_reset_conf**.

Data Structures

Request/Indication

The operation specific structure described below is used by the Client in issuing a Reset request (**mp_reset**). It is received by the Server when a Reset indication (**u_reset_ind**) is received.

```
struct reset_req_info
  {
  ST_CHAR piname [MAX_IDENT_LEN+1];
  };
typedef struct reset_req_info RESET_REQ_INFO;
```

Fields:

`piname` This contains the name of the program invocation to be reset.

Paired Primitive Interface Functions

mp_reset

Usage: This primitive request function sends a Reset request PDU using the data from a structure of type **RESET_REQ_INFO**, pointed to by **info**. The Reset service is used to cause the specified program invocation at the remote node to leave the **PI_STOPPED** or **PI_RUNNING** state and to enter the **PI_IDLE** state.

Function Prototype:

```
MMSREQ_PEND *mp_reset (ST_INT chan,
                       RESET_REQ_INFO *info);
```

Parameters:

chan	This is the channel number over which the Reset PDU is to be sent.
info	This pointer to an Operation-Specific data structure of type **RESET_REQ_INFO** contains information specific to the Reset PDU to be sent.

Return Value:

MMSREQ_PEND *	This pointer to the request control data structure of type **MMSREQ_PEND** is used to send the Reset PDU. In case of an error, the pointer is set to null and **mms_op_err** is written with the error code.

Corresponding User Confirmation Function: **u_mp_reset_conf**

Operation-Specific Data Structure Used: **RESET_REQ_INFO**

See page 2-419 for a detailed description of this structure.

u_reset_ind

Usage: This user indication function is called when a Reset indication is received by a Server node. The contents and operation of this function are user-defined and depend on the application requirements. To conform to the MMS protocol requirements, the application must do one of the following some time after the indication is received:

1) send a positive response using either:

   a) the primitive response function (**`mp_reset_resp`**) after the state of the specified program invocation has been successfully changed from the **`PI_STOPPED`** or **`PI_RUNNING`** state to the **`PI_IDLE`** state, or

   b) the virtual machine response function (**`mv_reset_resp`**) to have the virtual machine take the requested action and send the response, or

2) call the appropriate primitive negative (error) response function (**`mp_err_resp`**). See **Volume 3 — Module 11 — MMS-EASE Error Handling** for more information on this function.

See **Volume 1 — Module 1 — User Indication Function Class** for additional information on the recommended handling of indications.

Function Prototype: `ST_VOID u_reset_ind (MMSREQ_IND *ind);`

Parameters:

`ind` This pointer to the **`MMSREQ_IND`** structure is described in **Volume 1 — Module 2 — Request and Indication Control Data Structures**. This pointer must be used to send the response to this indication. The primary item of interest in this structure is **`ind->req_info_ptr`**. This is a pointer to the operation-specific data structure for a Reset indication (**`RESET_REQ_INFO`**). This pointer will always be valid when **`u_reset_ind`** is called.

Return Value: `ST_VOID` (ignored)

Operation-Specific Data Structure Used: **`RESET_REQ_INFO`**

See page 2-419 for a detailed description of this structure.

mp_reset_resp

Usage: This primitive response function sends a Reset positive response PDU. This should be called after the **u_reset_ind** function is called (a Reset indication was received), and after the state of the specified program invocation has been successfully changed from the **PI_STOPPED** or **PI_RUNNING** state to the **PI_IDLE** state.

Function Prototype: `ST_RET mp_reset_resp (MMSREQ_IND *ind);`

Parameters:

`ind` This pointer to the indication control structure of type **MMSREQ_IND** is passed to the **u_reset_ind** function when it was called.

Return Value: `ST_RET` **SD_SUCCESS**. No Error.

`<> 0` Error Code.

Corresponding User Indication Function: **u_reset_ind**

Operation-Specific Data Structure Used: NONE

u_mp_reset_conf

Usage: This primitive user confirmation function is called when a confirm to a Reset request (**mp_reset**) is received. **resp_err** contains a value indicating whether an error occurred. **resp_info_ptr** may contain data that the remote node sent back in the confirmation or it may contain error information if an error occurred. See **Volume 1 — Module 1 — Primitive User Confirmation Function Class** for more information on the recommended handling of confirms.

Note that the function called for this event may be changed if desired; however, **u_mp_reset_conf** is the standard default function.

Function Prototype:

```
ST_VOID u_mp_reset_conf (MMSREQ_PEND *req);
```

Parameters:

`req` This pointer to the **MMSREQ_PEND** structure is returned from the original **mp_reset** request function. See **Volume 1 — Module 2 — Request and Indication Control Data Structures** for more information on this structure.

For a positive response (**req->resp_err = CNF_RESP_OK**), the request control structure member **resp_info_pres** indicates that there is no confirm data for this function.

For a negative response, or other errors, see **Volume 3 — Module 11 — MMS-EASE Error Handling** for more information.

Return Value: `ST_VOID` (ignored)

Operation-Specific Data Structure Used: NONE

Virtual Machine Reset Operations

The following section contains information on how to use the virtual machine interface for the Reset service.

- The Reset service consists of the virtual machine function of **mv_reset_resp**.

There are no virtual machine level data structures.

Virtual Machine Interface Functions

mv_reset_resp

Usage: This virtual machine function allows the user to respond to a Reset response without having to create explicitly an input structure of type **RESET_RESP_INFO**. Before calling this function, the user must call the support function, **ms_check_pi_state**. This makes sure that the program invocation is in the correct state, and that the application program can actually perform this service. See page 2-383 for more information on this support function. This service returns a zero (0) if it is a SUCCESS and also updates the **PROG_INV** structure. If an error occurs, **mms_op_err** code is used if MMS-EASE could not send the response. See **Volume 1 — Appendix A** for more information on this code's possible values. MMS-EASE assigns all channels to reference the default **VMD_CTRL** structure when **strt_MMS()** is called. If more than one **VMD_CTRL** structure is being used, BE SURE TO SET **mms_chan_info.objs.vmd** TO POINT TO THE PROPER VMD STRUCTURE BEFORE CALLING THIS FUNCTION. See **Volume 1 — Module 2 — Channel Information Structure** and **Module 4 — VMD Control** starting on page 2-10 for more information.

Function Prototype: `ST_RET mv_reset_resp (MMSREQ_IND *ind, ST_RET err);`

Parameters:

`ind` This pointer to the indication control data structure of type **MMSREQ_IND** is passed to the **u_reset_ind** function when it was called.

`err` This is the value normally derived by calling the **ms_check_pi_state** function. When this argument is NOT equal to zero (<> 0), then this value indicates the reason for sending a negative response. The following errors are specific to Program Invocations:

PI_NAME	**200**	A Program Invocation Object by that name already exists or a Program Invocation Object should exist but it doesn't.
PI_STATE	**201**	A bad PI state is returned for a given operation.
PI_NOT_REUSABLE	**202**	Cannot reset this Program Invocation.
PI_NOT_DELETABLE	**203**	Cannot delete this Program Invocation.
PI_PROTECTION	**204**	Privilege is invalid to manage this Program Invocation.
INVALID_ID	**205**	Invalid MMS Identifier.

Return Value: `ST_RET` **SD_SUCCESS**. No Error.

`<> 0` Error Code.

26. Kill Service

This service is used to cause the state of a specified program invocation at the server to be placed in the **PI_UNRUNNABLE** state.

Primitive Level Kill Operations

The following section contains information on how to use the paired primitive interface for the Kill service. It covers available data structures used by the PPI, and the four primitive level functions that together make up the Kill service. See **Volume 1 — Module 2 — General Sequence of Events — Confirmed Services** for a detailed explanation of the sequence of events that occur within the MMS-EASE environment during confirmed services.

- The Kill service consists of the paired primitive functions of **mp_kill**, **u_kill_ind**, **mp_kill_resp**, and **u_mp_kill_conf**.

Data Structures

Request/Indication

The operation specific structure described below is used by the Client in issuing a Kill request (**mp_kill**). It is received by the Server when a Kill indication (**u_kill_ind**) is received.

```
struct kill_req_info
  {
  ST_CHAR piname [MAX_IDENT_LEN+1];
  };
typedef struct kill_req_info KILL_REQ_INFO;
```

Fields:

piname This contains the name of the program invocation to be killed.

Paired Primitive Interface Function

mp_kill

Usage: This primitive request function sends a Kill request PDU using the data from a structure of type **KILL_REQ_INFO**, pointed to by **info**. The Kill service is used to cause the specified program invocation at the remote node to leave its current state and to enter the **PI_UNRUNNABLE** state.

Function Prototype: `MMSREQ_PEND *mp_kill (ST_INT chan, KILL_REQ_INFO *info);`

Parameters:

`chan` This is the channel number over which the Kill PDU is to be sent.

`info` This pointer to an Operation-Specific data structure of type **KILL_REQ_INFO** contains information specific to the Kill PDU to be sent.

Return Value:

`MMSREQ_PEND *` This pointer to the request control data structure of type **MMSREQ_PEND** is used to send the Kill PDU. In case of an error, the pointer is set to null and **mms_op_err** is written with the error code.

Corresponding User Confirmation Function: **u_mp_kill_conf**

Operation-Specific Data Structure Used: **KILL_REQ_INFO**

See page 2-425 for a detailed description of this structure.

u_kill_ind

Usage: This user indication function is called when a Kill indication is received by a Server node. The contents and operation of this function are user-defined and depend on the application requirements. To conform to the MMS protocol requirements, the application must do one of the following some time after the indication is received:

1) send a positive response using either:

   a) the primitive response function (**mp_kill_resp**) after the specified program invocation has been successfully placed into the **PI_UNRUNNABLE** state, or

   b) the virtual machine response function (**mv_kill_resp**) to have the virtual machine take the requested action and send the response, or

2) call the appropriate primitive negative (error) response function (**mp_err_resp**). See **Volume 3 — Module 11 — MMS-EASE Error Handling** for more information on this function.

See **Volume 1 — Module 1 — User Indication Function Class** for additional information on the recommended handling of indications.

Function Prototype: `ST_VOID u_kill_ind (MMSREQ_IND *ind);`

Parameters:

`ind` This pointer to the **MMSREQ_IND** structure is described in **Volume 1 — Module 2 — Request and Indication Control Data Structures**. This pointer must be used to send the response to this indication. The primary item of interest in this structure is **ind->req_info_ptr**. This is a pointer to the operation-specific data structure for a Kill indication (**KILL_REQ_INFO**). This pointer will always be valid when **u_kill_ind** is called.

Return Value: `ST_VOID` (ignored)

Operation-Specific Data Structure Used: **KILL_REQ_INFO**

See page 2-425 for a detailed description of this structure.

mp_kill_resp

Usage: This primitive response function sends a Kill positive response PDU. It should be called after the **u_kill_ind** function is called (a Kill indication was received), and after the specified program invocation has been successfully placed into the **PI_UNRUNNABLE** state.

Function Prototype: `ST_RET mp_kill_resp (MMSREQ_IND *ind);`

Parameters:

`ind` This pointer to the indication control structure of type **MMSREQ_IND** is passed to the **u_kill_ind** function when it was called.

Return Value: `ST_RET` **SD_SUCCESS**. No Error.

`<> 0` Error Code.

Corresponding User Indication Function: **u_kill_ind**

Operation-Specific Data Structure Used: NONE

u_mp_kill_conf

Usage: This primitive user confirmation function is called when a confirm to a Kill request (**mp_kill**) is received. **resp_err** contains a value indicating whether an error occurred. **resp_info_ptr** may contain data that the remote node sent back in the confirmation or it may contain error information if an error occurred. See **Volume 1 — Module 1 — Primitive User Confirmation Function Class** for more information on the recommended handling of confirms.

Note that the function called for this event may be changed if desired; however, **u_mp_kill_conf** is the standard default function.

Function Prototype:

```
ST_VOID u_mp_kill_conf (MMSREQ_PEND *req);
```

Parameters:

`req` This pointer to the **MMSREQ_PEND** structure is returned from the original **mp_kill** request function. See **Volume 1 — Module 2 — Request and Indication Control Data Structures** for more information on this structure.

For a positive response (**req->resp_err = CNF_RESP_OK**), the request control structure member **resp_info_pres** indicates that there is no confirm data for this function.

For a negative response, or other errors, see **Volume 3 — Module 11 — MMS-EASE Error Handling** for more information.

Return Value: `ST_VOID` (ignored)

Operation-Specific Data Structure Used: NONE

Virtual Machine Kill Operations

The following section contains information on how to use the virtual machine interface for the Kill service.

- The Kill service consists of the virtual machine function of **mv_kill_resp**.

There are no virtual machine level data structures.

Virtual Machine Interface Functions

mv_kill_resp

Usage: This virtual machine function allows the user to respond to a Kill response without having to create explicitly an input structure of type **KILL_RESP_INFO**. Before calling this function, the user must call the support function, **ms_check_pi_state**. This makes sure that the program invocation is in the correct state, and that the application program can actually perform this service. See page 2-383 for more information on this support function. It returns a zero (0) if it is a SUCCESS and also updates the **PROG_INV** structure. If an error occurs, **mms_op_err** code is used if MMS-EASE could not send the response. See **Volume 1 — Appendix A** for more information on this code's possible values. MMS-EASE assigns all channels to reference the default **VMD_CTRL** structure when **strt_MMS()** is called. If more than one **VMD_CTRL** structure is being used, BE SURE TO SET **mms_chan_info.objs.vmd** TO POINT TO THE PROPER VMD STRUCTURE BEFORE CALLING THIS FUNCTION. See **Volume 1 — Module 2 — Channel Information Structure** and **Module 4 — VMD Control** starting on page 2-10 for more information.

Function Prototype: `ST_RET mv_kill_resp (MMSREQ_IND *ind, ST_RET err);`

Parameters:

`ind` This pointer to the indication control data structure of type **MMSREQ_IND** is passed to the **u_kill_ind** function when it was called.

`err` This is the value normally derived by calling the **ms_check_pi_state** function. When this argument is NOT equal to zero (<> 0), then this value indicates the reason for sending a negative response. The following errors are specific to Program Invocations:

PI_NAME	**200**	A Program Invocation Object by that name already exists or a Program Invocation Object should exist but it doesn't.
PI_STATE	**201**	A bad PI state is returned for a given operation.
PI_NOT_REUSABLE	**202**	Cannot reset this Program Invocation.
PI_NOT_DELETABLE	**203**	Cannot delete this Program Invocation.
PI_PROTECTION	**204**	Privilege is invalid to manage this Program Invocation.
INVALID_ID	**205**	Invalid MMS Identifier.

Return Value: `ST_RET` **SD_SUCCESS**. No Error.

`<> 0` Error Code.

27. GetProgramInvocationAttributes Service

This service is used to request that all attributes associated with a specified program invocation be returned from the Server.

Primitive Level GetProgramInvocationAttributes Operations

The following section contains information on how to use the paired primitive interface for the GetProgramInvocationAttributes service. It covers available data structures used by the PPI, and the four primitive level functions that together make up the GetProgramInvocationAttributes service. See **Volume 1 — Module 2 — General Sequence of Events — Confirmed Services** for a detailed explanation of the sequence of events that occur within the MMS-EASE environment during confirmed services.

- The GetProgramInvocationAttributes service consists of the paired primitive functions of **`mp_getpi`**, **`u_getpi_ind`**, **`mp_getpi_resp`**, and **`u_mp_getpi_conf`**.

Data Structures

Request/Indication

The operation specific structure described below is used by the Client in issuing a GetProgramInvocationAttributes request (**`mp_getpi`**). It is received by the Server when a GetProgramInvocationAttributes indication (**`u_getpi_ind`**) is received.

```
struct getpi_req_info
  {
  ST_CHAR piname [MAX_IDENT_LEN+1];
  };
typedef struct getpi_req_info GETPI_REQ_INFO;
```

Fields:

`piname` This contains the name of the program invocation for which the attributes are being requested.

Response/Confirm

The operation specific data structure described below is used by the Server in issuing a GetProgramInvocationAttributes response (**`mp_getpi_resp`**). It is received by the Client when a GetProgramInvocationAttributes confirm (**`u_mp_getpi_conf`**) is received.

```
struct getpi_resp_info
  {
  ST_INT16    state;
  ST_BOOLEAN mms_deletable;
  ST_BOOLEAN reusable;
  ST_BOOLEAN monitor;
  ST_INT16    start_arg_type;
  ST_INT      start_arg_len;
  ST_UCHAR    *start_arg;
  ST_INT      num_of_dnames;
/*ST_CHAR     *dnames_list [num_of_dnames];                        */
  SD_END_STRUCT
  };
typedef struct getpi_resp_info GETPI_RESP_INFO;
```

Fields:

`state` This indicates the states of the program invocation:

PI_NON_EXISTENT
PI_UNRUNNABLE
PI_IDLE
PI_RUNNING
PI_STOPPED
PI_STARTING
PI_STOPPING
PI_RESUMING
PI_RESETTING

`mms_deletable` **SD_FALSE**. The program invocation cannot be deleted using a MMS service request.

SD_TRUE. The program invocation can be deleted using a MMS service request.

`reusable` **SD_FALSE**. This program invocation is not reusable, and goes to the **PI_UNRUNNABLE** state following completion of execution.

SD_TRUE. This program invocation is reusable, and goes to the **PI_IDLE** state following completion of execution.

`monitor` **SD_FALSE**. Current monitoring. Program monitoring is NOT in effect.

SD_TRUE. Permanent monitoring. This program invocation uses the event management services (GetEventEnrollment service) to inform an external node when the program invocation leaves the **PI_RUNNING** state.

`start_arg_type` This indicates the type of argument used to pass data to the program invocation:

SIMPLE_STRING. This is a null-terminated character string.

ENCODED_STRING. This is a value encoded by some other external source - ASN.1 EXTERNAL data element. See page 2-201 for more information on how to encode and decode ASN.1 EXTERNAL data elements.

`start_arg_len` This length, in bytes, of the data present in **start_arg** is used only if **start_arg_type = ENCODED_STRING**.

`start_arg` This pointer to the start arguments is passed to this program invocation during the most recent Start request.

`num_of_dnames` This indicates the number of elements in the **dname_list** array.

`dnames_list` This array of pointers points to the names of the domains bound to this program invocation.

NOTE: FOR RESPONSE ONLY, when allocating a data structure of type **GETPI_RESP_INFO**, enough memory must be allocated to hold the information for the **dnames_list** members of the structure. The following C statement can be used:

```
info = (GETPI_RESP_INFO *) chk_malloc (sizeof (GETPI_RESP_INFO) + num_of_dnames *
            sizeof(ST_CHAR *));
```

Paired Primitive Interface Functions

mp_getpi

Usage: This primitive request function sends a GetProgramInvocationAttributes request PDU using the data from a structure of type **GETPI_REQ_INFO**, pointed to by **info**. This service is used to obtain a list of the attributes of a particular program invocation at the remote node.

Function Prototype:

```
MMSREQ_PEND *mp_getpi (ST_INT chan,
                       GETPI_REQ_INFO *info);
```

Parameters:

`chan` This is the channel number over which the GetProgramInvocationAttributes PDU is to be sent.

`info` This pointer to an Operation-Specific data structure of type **GETPI_REQ_INFO** contains information specific to the request PDU to be sent.

Return Value:

`MMSREQ_PEND *` This pointer to the request control data structure of type **MMSREQ_PEND** is used to send the PDU. In case of an error, the pointer is set to null and **mms_op_err** is written with the error code.

Corresponding User Confirmation Function: **u_mp_getpi_conf**

Operation-Specific Data Structure Used: **GETPI_REQ_INFO**

See page 2-431 for a detailed description of this structure.

u_getpi_ind

Usage: This user indication function is called when a GetProgramInvocationAttributes indication is received by a Server node. The contents and operation of this function are user-defined and depend on the application requirements. To conform to the MMS protocol requirements, the application must do one of the following some time after the indication is received:

1) send a positive response using either:

   a) the primitive response function (**mp_getpi_resp**) after the specified program invocation attribute information has been successfully obtained, or

   b) the virtual machine response function (**mv_getpi_resp**) to have the virtual machine take the requested action and send the response, or

2) call the appropriate primitive negative (error) response function (**mp_err_resp**). See **Volume 3 — Module 11 — MMS-EASE Error Handling** for more information on this function.

See **Volume 1 — Module 1 — User Indication Function Class** for additional information on the recommended handling of indications.

Function Prototype:

```
ST_VOID u_getpi_ind  (MMSREQ_IND *ind);
```

Parameters:

`ind` This pointer to the **MMSREQ_IND** structure is described in **Volume 1 — Module 2 — Request and Indication Control Data Structures**. This pointer must be used to send the response to this indication. The primary item of interest in this structure is **ind->req_info_ptr**. This is a pointer to the operation-specific data structure for a GetProgramInvocationAttributes indication (**GETPI_REQ_INFO**). This pointer will always be valid when **u_getpi_ind** will be called.

Return Value: `ST_VOID` (ignored)

Operation-Specific Data Structure Used: **GETPI_REQ_INFO**

See page 2-431 for a detailed description of this structure.

mp_getpi_resp

Usage: This primitive response function sends a GetProgramInvocationAttributes positive response PDU. This function should be called after the **u_getpi_ind** function is called (a GetProgramInvocationAttributes indication was received), and after the specified program invocation attribute information has been successfully obtained.

Function Prototype:

```
ST_RET mp_getpi_resp (MMSREQ_IND *ind,
                      GETPI_RESP_INFO *info);
```

Parameters:

`ind` This pointer to the indication control structure of type **MMSREQ_IND** is passed to the **u_getpi_ind** function when it was called.

`info` This pointer to an Operation-Specific data structure of type **GETPI_RESP_INFO** contains information specific to the response PDU to be sent.

Return Value: `ST_RET` **SD_SUCCESS**. No Error.

`<> 0` Error Code.

Corresponding User Indication Function: **u_getpi_ind**

Operation-Specific Data Structure Used: **GETPI_RESP_INFO**

See page 2-431 for a detailed description of this structure.

u_mp_getpi_conf

Usage: This primitive user confirmation function is called when a confirm to a GetProgramInvocationAttributes request (**mp_getpi**) is received. **resp_err** contains a value indicating whether an error occurred. **resp_info_ptr** may contain data that the remote node sent back in the confirmation or it may contain error information if an error occurred. If no error has occurred (**req->resp_err = CNF_RESP_OK**), GetProgramInvocationAttributes information is available to the application using the **req_ptr->resp_info_ptr** pointer. See **Volume 1 — Module 1 — Primitive User Confirmation Function Class** for more information on the recommended handling of confirms.

Note that the function called for this event may be changed if desired; however, **u_mp_getpi_conf** is the standard default function.

Function Prototype:

```
ST_VOID u_mp_getpi_conf (MMSREQ_PEND *req);
```

Parameters:

`req` This pointer to the **MMSREQ_PEND** structure is returned from the original **mp_getpi** request function. See **Volume 1 — Module 2 — Request and Indication Control Data Structures** for more information on this structure.

For a positive response (**req->resp_err = CNF_RESP_OK**), the request control structure member **resp_info_ptr** is a pointer to the operation-specific data structure (**GETPI_RESP_INFO**) for the GetProgramInvocationAttributes function.

For a negative response, or other errors, see **Volume 3 — Module 11 — MMS-EASE Error Handling** for more information.

Return Value: `ST_VOID` (ignored)

Operation-Specific Data Structure Used: **GETPI_RESP_INFO**

See page 2-431 for a detailed description of this structure.

Virtual Machine GetProgramInvocationAttributes Operations

The following section contains information on how to use the virtual machine interface for the GetProgramInvocationAttributes service.

- The GetProgramInvocationAttributes service consists of the virtual machine response function, **mv_getpi_resp**.

There are no virtual machine level data structures.

Virtual Machine Interface Functions

mv_getpi_resp

Usage: This virtual machine function allows the user to respond to a Get Program Invocation Attributes response without having to create explicitly an input structure of type **GETPI_RESP_INFO**. Before calling this function, the user must call the support function, **ms_check_pi_state**. This makes sure that the remote application has the appropriate privileges to request this particular program invocation's attributes. See page 2-383 for more information on this support function. It returns a zero (0) if it is a SUCCESS and also updates the **PROG_INV** structure. If an error occurs, **mms_op_err** code is used if MMS-EASE could not send the response. See **Volume 1 — Appendix A** for more information on this code's possible values. MMS-EASE assigns all channels to reference the default **VMD_CTRL** structure when **strt_MMS()** is called. If more than one **VMD_CTRL** structure is being used, BE SURE TO SET **mms_chan_info.objs.vmd** TO POINT TO THE PROPER VMD STRUCTURE BEFORE CALLING THIS FUNCTION. See **Volume 1 — Module 2 — Channel Information Structure** and **Module 4 — VMD Control** starting on page 2-10 for more information.

Function Prototype: `ST_RET mv_getpi_resp (MMSREQ_IND *ind, ST_RET err);`

Parameters:

`ind` This pointer to the request data structure of type **MMSREQ_IND** is passed to the **u_getpi_ind** function when it was called.

`err` This is the value normally derived by calling the **ms_check_pi_state** function. When this argument is NOT equal to zero (`<> 0`), then this value indicates the reason for sending a negative response. The following errors are specific to Program Invocations:

PI_NAME	**200**	A Program Invocation Object by that name already exists or a Program Invocation Object should exist but it doesn't.
PI_STATE	**201**	A bad PI state is returned for a given operation.
PI_NOT_REUSABLE	**202**	Cannot reset this Program Invocation.
PI_NOT_DELETABLE	**203**	Cannot delete this Program Invocation.
PI_PROTECTION	**204**	Privilege is invalid to manage this Program Invocation.
INVALID_ID	**205**	Invalid MMS Identifier.

Return Value: `ST_RET` **SD_SUCCESS**. No Error.

`<> 0` Error Code.

1. Modifier Handling

Every confirmed MMS service contains a “List of Modifier” parameter. This is an option in the request and indication primitives. This parameter serves to specify a list of one or more service state machine modifiers. These add a condition that must be satisfied for the execution of the service request to begin.

MMS defines two modifiers explicitly:

1. Attach to Semaphore modifier. See page 3-59 for more information.
2. Attach to Event Condition modifier. See page 3-197 for more information.

Modifier Structures

The following structures are used to support modifier handling in a Confirmed-Request PDU. For more specific information on the specific modifiers (**ATTACH_TO_SEMAPHORE** and **ATTACH_TO_EVCON**) that are used by MMS-EASE, please refer to pages 3-59 and 3-197 for more information.

```
struct list_of_mods
  {
  ST_BOOLEAN info_pres;
  ST_INT      num_of_mods;
  struct      modifier     *mod_list_ptr;
  };
typedef struct list_of_mods LIST_OF_MODS;
```

Fields:

`info_pres` **SD_FALSE**. Do NOT allow modifier information to be encoded into the PDU.

SD_TRUE. Allow modifier information to be encoded into the PDU. This must be set in order for the modifier information to be encoded.

`num_of_mods` This indicates the number of the modifier structures in the modifier list.

`mod_list_ptr` This pointer of structure **MODIFIER** points to a contiguous array of modifiers.

```
extern struct list_of_mods modifier_list;
```

This global variable serves as the mechanism to inform the MMS encode state machine that it must encode a modifier. **modifier_list** must be assigned correct values, and the **modifier_list.info_pres** flag must be set to TRUE before the next primitive level or virtual machine request function is called. This variable is only good for sending one modifier per request as MMS-EASE clears the **modifier_list.info_pres** flag after it encodes the modifier. This approach is similar to the approach used to send additional error information. In addition, the following structure needs to be defined in order to use the **Attach To Semaphore** or **Attach to Event Condition** modifier.

```
struct modifier
  {
  ST_INT modifier_tag;
  union
    {
    ATTACH_TO_EVCON     atec;
    ATTACH_TO_SEMAPHORE atsem;
    } mod;
  SD_END_STRUCT
  };
typedef struct modifier   MODIFIER;
```

Fields:

`modifier_tag` This indicates the type of modifier to be used. Options are:

- **`0`** This selects the **`attach_to_evcon`** modifier.
- **`1`** This selects the **`attach_to_semaphore`** modifier.

`atec` This structure of type **`ATTACH_TO_EVCON`** contains the attach to event condition modifier to be used. It is used if **`modifier_tag = 0`**. See page 3-197 for a detailed description of this structure.

`atsem` This structure of type **`ATTACH_TO_SEMAPHORE`** contains the attach to semaphore modifier to be used. It is used if **`modifier_tag = 1`**. See page 3-59 for a detailed description of this structure.

Using Modifiers

Sending a Modifier in a Confirmed Request

To send a modifier in a confirmed request PDU, the global variable **`modifier_list`** is used. This serves as a mechanism to inform the MMS Encode State Machine that it must encode a modifier. **`modifier_list`** must be assigned correct values, and the **`modifier_list.info_pres`** flag must be set to **`SD_TRUE`** before the next primitive level or virtual machine request function is called. This variable is only good for sending one modifier per request as MMS-EASE clears the **`modifier_list.info_pres`** flag after it encodes the modifier. In addition, the **`MMSREQ_PEND`** structure was modified to include the addition of the **`mods`** member. This field contains information about the modifier that was sent out in the request. This field is strictly for application use. It is used with the intent that some time later, the application will be presented in a confirmation function with a pointer to the original request information. It may be helpful to examine the modifier at that point. See **Volume 1 — Module 2 — General — Request and Indication Control Data Structures (`MMSREQ_PEND`)** for additional information.

Receiving a Modifier from a Confirmed Request

The modifier is decoded from the PDU and presented in the **`MMSREQ_IND`** structure. The **`MMSREQ_IND`** structure has been modified by the addition of the **`mods`** member. The **`mods.info_pres`** flag will be set to indicate the presence of a modifier. The application should not alter any of the **`mod`** values as unpredictable results may occur. See **Volume 1 — Module 2 — General — Request and Indication Control Data Structures (`MMSREQ_IND`)** for additional information.

In addition, the following configuration variable is important in receiving a modifier:

```
extern ST_INT m_max_mods;
```

This variable contains the maximum number of modifiers present in the **`modifier_list`** structure that the responder is willing to decode. The default is set to 1. See page 3-1 for more information on modifier handling. If more than this number of modifiers present is used, it will not be parsed, and a reject will be generated. This variable is present so that the application program can tune memory usage.

Sending a Modifier Position in a Confirmed Error PDU

The application program would normally examine the **`modifier_list`** structure and take some action based on the results. There is no VMD support for modifiers. If exception is taken to a modifier in the list (i.e., because the semaphore being referenced does not exist), an error can be sent back. Part of every confirmed error PDU is the modifier position (**`mod_pos`**). See **Module 11 — MMS-EASE Error Handling** starting on page 339 for more information on this member. Sending this member assigns **`mod_pos`** to the modifier position and sets the modifier position flag to TRUE (**`mod_pos_pres = 0`**). The next call to **`mp_err_resp`** encodes this field and clears the flag.

Receiving a Modifier Position in a Confirmed Error PDU

When an application program receives a Confirmed-Error PDU, it may examine the **`err_info`** structure. See **Module 11 — MMS-EASE Error Handling** starting on page 3-339 for a detailed description of this structure.

2. Common Data Structure

The following structure holds the Binary Time Of Day information used in several Operation-Specific support structures regarding events, journals and alarms.

```
typedef struct btod_data
  {
  ST_INT   form;
  ST_INT32 ms;
  ST_INT32 day;
  } MMS_BTOD;
```

Fields:

`form` This indicates which form of the Binary Time of Data to use:

`MMS_BTOD4`. This means use **Btime4** which is encoded as four octets.

`MMS_BTOD6`. This means use **Btime6** which is encoded as six octets.

`ms` This contains the number of milliseconds since 0:00 (midnight) January 1, 1970.

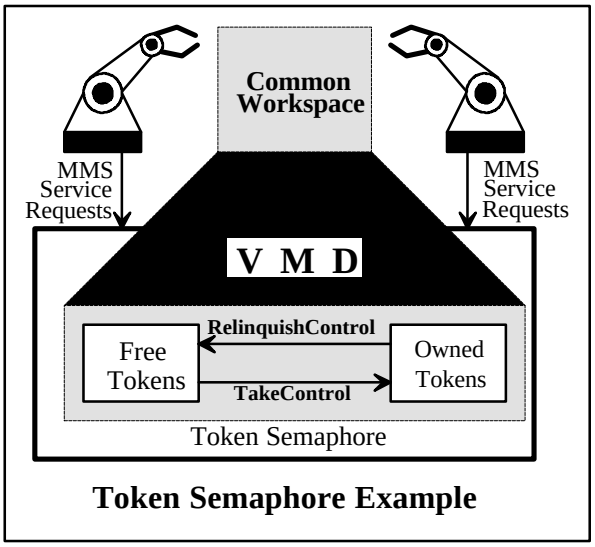
`day` This contains the number of days since January 1, 1984.

3. Semaphore Management Introduction

This portion of MMS-EASE provides services that allow multiple MMS-users to share common resources represented by semaphores. In many real-time systems, there is a need for a mechanism by which an application can control access to a system resource. An example might be a workspace that is physically accessible to several robots. Some means to control which robot (or robots) can access the workspace is needed. MMS defines two types of *semaphores* for these types of applications.

Token Semaphores

A token semaphore is a named MMS object that can be a representation of some resource within the control of the VMD to which access must be controlled. A token semaphore is modeled as a collection of tokens that MMS clients take and relinquish control of using MMS services. This allows both multiple or exclusive ownership of the semaphore. When a MMS Client owns the token, it provides some level of access to the underlying resource. An example might be where two users what to change the setpoint for the same control loop at the same time. These users could use a MMS token semaphore containing only one token to represent the control loop in order to coordinate their access to the setpoint. When the user "owns" the token, they can change the setpoint. The other would have to wait until ownership is relinquished.



Token Semaphore Example

A token semaphore can also be used for the sole purpose of coordinating the activities of two MMS clients without representing any real resource. This kind of "virtual" token semaphore looks and behaves the same except that they can be created and deleted by MMS clients using DefineSemaphore service.

Because semaphores either represent a real resource or are used for the purpose of coordinating activities between two or more MMS clients, the scope of a semaphore cannot be AA-specific. If an object's scope is AA-specific there can only be one client. Also, AA-specific objects only exist as long as the application association exists while real resources must exist outside the scope of any given application association.

A token semaphore is modeled as a collection of free tokens and owned tokens. When a robot (a MMS client in this example) wants to access the common workspace, it will issue a TakeControl request to the VMD controlling the workspace. If there is a free token available, the VMD will grant control by moving a token from the FREE state to the OWNED state and then responding positively to the Take Control request. If the other robot had already owned the token, then the VMD would respond negatively to the TakeControl request. The token representing the common workspace would remain under the control of the robot until control was released with a RelinquishControl request or upon a timeout by the VMD. The total number of tokens available indicates how many simultaneous owners can exist for the same semaphore.

In addition to its name, the token semaphore has the following attributes:

Deletable If True, it means that the semaphore does not represent any real resource within the VMD and can therefore be deleted by a MMS client using the DeleteSemaphore service.

Number of Tokens This attribute indicates the total number of tokens contained in the token semaphore.

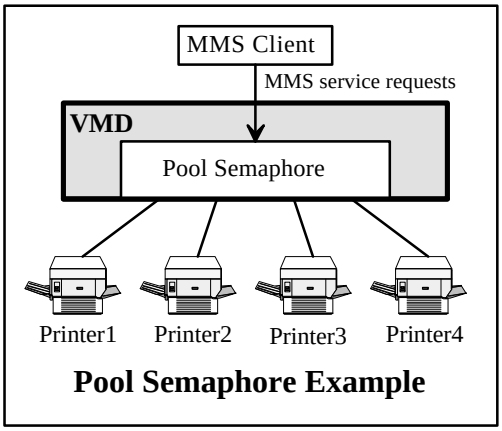
Owned Tokens This attribute indicates the number of tokens whose associated semaphore entry state is OWNED.

Hung Tokens This attribute indicates the number of tokens whose associated semaphore entry state is HUNG.

Pool Semaphores

A pool semaphore is similar to a token semaphore except that the individual tokens are identifiable and have a name associated with them. These *named tokens* can optionally be specified by the MMS client when issuing TakeControl requests. The pool semaphore itself is a MMS object. The named tokens contained in the pool semaphore are not MMS objects. They are representations of a real resource in much the same way an unnamed variable object is a representation of a real resource. The name of the pool semaphore is separate from the names of the named tokens. Pool semaphores can only be used represent some real resource within the VMD. Therefore, pool semaphores cannot be created or deleted using MMS service requests and cannot be AA-specific in scope.

In addition to the name of the pool semaphore, the following attributes are also defined by MMS:



A pool semaphore can be useful to control access to similar but distinguishable resources. In the example above each printer is represented as a separate named token. A MMS client can request control of a specific printer by issuing a TakeControl request specifying a named token. Alternately, if the client doe s not care which specific printer it is granted control of, it can issue the TakeControl request without specifying a named token.

Figure 3-1: Pool Semaphore Example

Free Named Tokens The named tokens for which no associated semaphore entry exists.

Owned Named Tokens The named tokens for which associated semaphore entry state is OWNED.

Hung Named Tokens The named tokens for which the associated semaphore entry is HUNG.

Semaphore Entry

When a MMS client issues a TakeControl request for a given semaphore, the VMD creates an entry in an internal queue that is maintained for each semaphore. Each entry in this queue is called a *semaphore entry*. The attributes of a semaphore entry are visible to MMS clients and provide information about the internal semaphore processing queue in the VMD. The semaphore entry is not a MMS object. It only exists from the receipt of the TakeControl indication by the VMD until the token control of the semaphore is relinquished or if the VMD responds negatively to the TakeControl request. Several of the attributes of the semaphore entry are specified by the client in the TakeControl request. These attributes are:

Entry ID This is a number assigned by the VMD to distinguish one semaphore entry from another. Each Entry ID is unique to a given semaphore.

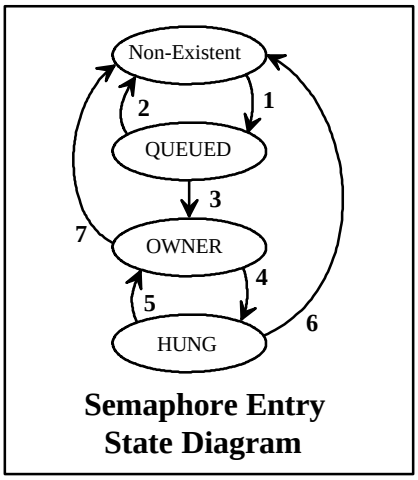
Named Token Valid only for pool semaphores. It contains the named token that was optionally requested by the client in a TakeControl request, or if the semaphore entry is in the OWNED, or HUNG state, it is the named token that the VMD assigned as a result of a TakeControl request.

Application Reference This is a reference to the MMS client application that issued the TakeControl request that created the semaphore entry.

Priority This attribute indicates the priority of the semaphore entry with respect to the other semaphore entries. Priority is used to decide which semaphore entry in the QUEUED state will be granted a token (or named token) when multiple requests are outstanding. The value (0=highest priority, 64=normal priority, 127=lowest priority) is specified by the client in the TakeControl request.

Entry State The entry state attributes represents the relationship between the MMS client and the semaphore by one of the following values:

- **QUEUED**. This means that a TakeControl request has been received but has not yet been responded to by the VMD. The client is waiting for control of the semaphore.
- **OWNED**. The VMS has responded positively to the TakeControl request and the client now owns the token (or named token).
- **HUNG**. This state means that the application association over which the MMS client issued the TakeControl request has been lost and the Relinquish is Connection Lost attribute is FALSE. A MMS client can take control of a semaphore entry in the HUNG state by issuing a TakeControl request with the *preempt* option TRUE and by specifying the MMS client to preempt (using the application reference attribute).



A semaphore entry is created each time a client attempts to take control of a semaphore. The semaphore entry reflects the state of the relationship between the client and the semaphore. State transitions can be caused by local autonomous action by the VMD or the following events:

1. *TakeControl request received.*
2. *Timeout, Cancel request or abort.*
3. *Semaphore available (token free).*
4. *Application association aborted and Rel.if.Conn.Lost = FALSE.*
5. *TakeControl with preempt.*
6. *Control timeout.*
7. *a) RelinquishControl request received, or b) control timeout, or c) application association aborted and Rel.if.Conn.Lost = TRUE.*

Figure 3-2: Semaphore Entry State Diagram

Relinquish if Connection Lost If this attribute is TRUE, the VMD will relinquish control of the semaphore if the application association for the MMS client that owned the token for this semaphore entry is lost or aborted.

Control Timeout This attribute is specified by the client in the TakeControl request and indicates how many milliseconds the client should be allowed to control the semaphore once control is granted. If the client has not relinquished control using a RelinquishControl request when the control timeout expires, the semaphore entry will be deleted and control of the semaphore will be relinquished. If the control timeout attribute is omitted in the TakeControl request, then no control timeout will apply for that semaphore entry.

Abort on Timeout This attribute is specified by the client in the TakeControl request and indicates if the VMD should abort the application association with the owner client upon a control timeout.

Acceptable Delay This attribute is specified by the client in the TakeControl request and indicates how many milliseconds the client is willing to wait for control of the semaphore. If control is not granted during this time, the VMD will respond negatively to the TakeControl request. If the acceptable delay attribute is omitted from the TakeControl request then the client is willing to wait indefinitely.

There are seven services used to support Semaphore Management:

TakeControl This service is used by a client to request control of a semaphore. See page 3-11 for more information.

RelinquishControl This service is used by a client to release control of a semaphore of which a client currently has control. See page 3-19 for more information.

DefineSemaphore This service is used by a client to define a token semaphore used solely for coordinating the activities of two or more clients. See page 3-25 for more information.

DeleteSemaphore This service is used by a client to delete a token semaphore used solely for coordinating the activities of two or more clients. See page 3-31 for more information.

ReportSemaphoreStatus This service is used to obtain the status (number of total, owned, and hung tokens) of a token semaphore. See page 3-37 for more information.

ReportPoolSemaphoreStatus This service is used to obtain the status (number of total, owned, and hung tokens) of a pool semaphore. See page 3-43 for more information.

ReportSemaphoreEntryStatus This service is used to obtain the attributes of semaphore entries corresponding to a specified state. See page 3-51 for more information.

The **ATTACH_TO_SEMAPHORE** modifier is also used to request that a server execute the service procedure associated with a specific service request under the control of a semaphore. See page 3-59 for information on modifier handling.

4. TakeControl Service

This service is used to request control of a semaphore. Receipt of a TakeControl indication at a server will create a semaphore entry in which the service parameters are recorded.

NOTE: It is up to the user's application program at the server to create the semaphore entry, and manage the queue. MMS-EASE does not do this.

Primitive Level TakeControl Operations

The following section contains information on how to use the paired primitive interface for the TakeControl service. It covers available data structures used by the PPI, and the four primitive level functions that together make up the TakeControl Service. **See Volume 1 — Module 2 — General Sequence of Events — Confirmed Services** for a detailed explanation of the sequence of events that occur within the MMS-EASE environment during confirmed services.

- The TakeControl service consists of the paired primitive functions of **`mp_takectrl`**, **`u_takectrl_ind`**, **`mp_takectrl_resp`**, and **`u_mp_takectrl_conf`**.

Data Structures

Request/Indication

The operation-specific structure described below is used by the client in issuing a TakeControl request (**`mp_takectrl`**). It is received by the server when a TakeControl indication (**`u_takectrl_ind`**) is received.

```
struct takectrl_req_info
  {
  OBJECT_NAME sem_name;
  ST_BOOLEAN  named_token_pres;
  ST_CHAR     named_token[MAX_IDENT_LEN+1];
  ST_UCHAR    priority;
  ST_BOOLEAN  acc_delay_pres;
  ST_UINT32   acc_delay;
  ST_BOOLEAN  ctrl_timeout_pres;
  ST_UINT32   ctrl_timeout;
  ST_BOOLEAN  abrt_on_timeout_pres;
  ST_BOOLEAN  abrt_on_timeout;
  ST_BOOLEAN  rel_conn_lost;
  ST_BOOLEAN  app_preempt_pres;
  ST_INT      app_len;
  ST_UCHAR    *app_preempt;
  SD_END_STRUCT
  };
typedef struct takectrl_req_info TAKECTRL_REQ_INFO;
```

Fields:

`sem_name` This contains the name of the semaphore of structure type **`OBJECT_NAME`** to be controlled. Semaphores may only be VMD-specific or domain-specific in scope.

`named_token_pres` **`SD_FALSE`**. Do Not include **`named_token`** in the PDU.

`SD_TRUE`. Include **`named_token`** in the PDU.

`named_token` This contains the named token of the semaphore if the semaphore is a pooled semaphore.

`priority` This specifies the priority that this TakeControl request should have concerning other requests for control of the same semaphore. Use a value of 0 for the highest priority, and 127 for the lowest priority. The default is 64 (normal).

`acc_delay_pres` **SD_FALSE**. Do Not include **acc_delay** in the PDU.

SD_TRUE. Include **acc_delay** in the PDU.

`acc_delay` This indicates the number of milliseconds of acceptable delay the remote node should wait for control of the semaphore before sending an error response. The default is no value indicating to wait indefinitely. If zero is indicated, no delay is acceptable meaning if the semaphore or named token is not available immediately, an error is received.

`ctrl_timeout_pres` **SD_FALSE**. Do Not include **ctrl_timeout** in the PDU.

SD_TRUE. Include **ctrl_timeout** in the PDU.

`ctrl_timeout` This indicates the number of milliseconds for which control should be granted. If the amount of time that control is held exceeds this value, the actions taken depend on the values of the **rel_conn_lost** and **abort_on_timeout** parameters. If not included, control should be granted indefinitely.

`abrt_on_timeout_pres` **SD_TRUE**. Include **abrt_on_timeout** in the PDU.

NOTE: Do not include **abrt_on_timeout** field if you do not include **ctrl_timeout**.

SD_FALSE. Do Not include **abrt_on_timeout** in the PDU.

NOTE: Must include **abort_on_timeout** if including **ctrl_timeout**.

`abrt_on_timeout` **SD_FALSE**. Do not abort the association when the control timeout, **ctrl_timeout**, is exceeded.

SD_TRUE. Abort the association when the control timeout, **ctrl_timeout** is exceeded.

`rel_conn_lost` **SD_TRUE**. Automatically relinquish control of the semaphore when the association that control was granted over is lost. This includes the case where the association is aborted because of **ctrl_timeout** and **abort_on_timeout != SD_FALSE**.

SD_FALSE. Do not automatically relinquish control of the semaphore when the association that control was granted over is lost. Instead the semaphore should remain OWNED and the corresponding semaphore entry should be HUNG.

`acc_preempt_pres` **SD_FALSE**. Do Not include **app_preempt** in the PDU.

SD_TRUE. Include **app_preempt** in the PDU.

`app_len` This indicates the length, in bytes, of **app_preempt**.

`app_preempt` This pointer to the ASN.1 encoded object identifier represents the AP Title of the application that is the owner of a HUNG semaphore for which control is to be preempted.

NOTE: See the description of **OBJECT_NAME** in **Volume 1 — Module 2 — MMS Object Name Structure** for more information on this structure.

Response/Confirm

The operation-specific data structure described below is used by the server in issuing a TakeControl response (**mp_takectrl_resp**). It is received by the client when a TakeControl confirm (**u_mp_takectrl_conf**) is received.

```
struct takectrl_resp_info
  {
  ST_INT16 resp_tag;
  ST_CHAR  named_token [MAX_IDENT_LEN+1];
  SD_END_STRUCT
  };
typedef struct takectrl_resp_info TAKECTRL_RESP_INFO;
```

Fields:

`resp_tag` This is a tag indicating what type of semaphore control was granted for:

- `0` The semaphore, for which control was granted, is a Token semaphore. No further information is needed. This is a null response.
- `1` The semaphore, for which control was granted, is a pool semaphore. Must use the **named_token** parameter.

`named_token` This contains the name of the specific Named Token for which control was granted if the semaphore is a pool semaphore (**resp_tag = 1**).

Paired Primitive Interface Functions

mp_takectrl

Usage: This primitive request function sends a TakeControl request PDU using the data from a structure of type **TAKECTRL_REQ_INFO**, pointed to by **info**. The TakeControl service is used to take control over a semaphore at a remote node.

Function Prototype:

```
MMSREQ_PEND *mp_takectrl (ST_INT chan,
                          TAKECTRL_REQ_INFO *info);
```

Parameters:

`chan` This integer contains the channel number over which the TakeControl PDU is to be sent.

`info` This pointer to the Operation-Specific data structure of type **TAKECTRL_REQ_INFO** contains information specific to the request PDU to be sent.

Return Value:

`MMSREQ_PEND *` This pointer to the queue data structure of type **MMSREQ_PEND** sends the PDU. In case of an error, the pointer is set to null and **mms_op_err** is written with the error code.

Corresponding User Confirmation Function: **u_mp_takectrl_conf**

Operation-Specific Data Structure Used: **TAKECTRL_REQ_INFO**

See page 3-11 for a detailed description of this structure.

u_takectrl_ind

Usage: This user indication function is called anytime a TakeControl indication is received by a server node. The contents and operation of this function are user-defined, and depend on the application requirements. To conform to the MMS protocol requirements, the application must do one of the following some time after the indication is received:

1) send a positive response using the primitive Response function (**`mp_takectrl_resp`**) after control of the specified semaphore has been successfully granted to the requesting node, or

2) call the appropriate primitive negative (error) response function (**`mp_err_resp`**). See **Module 11 — MMS-EASE Error Handling** on page 3-343 for an explanation of this function.

See **Volume 1 — Module 1 — User Indication Function Class** for additional information regarding the recommended handling of indications.

Function Prototype: `ST_VOID u_takectrl_ind (MMSREQ_IND *ind);`

Parameters:

`ind` This pointer to the **`MMSREQ_IND`** structure is described in **Volume 1 — Module 2 — Request and Indication Control Data Structures**. It must be used to send the response to this indication. The primary item of interest in this structure is **`ind->req_info_ptr`**. This is a pointer to the operation-specific data structure for a TakeControl indication (**`TAKECTRL_REQ_INFO`**). This pointer will always be valid when **`u_takectrl_ind`** is called.

Return Value: `ST_VOID` (ignored)

Operation-Specific Data Structure Used: **`TAKECTRL_REQ_INFO`**

See page 3-11 for a detailed description of this structure.

mp_takectrl_resp

Usage: This primitive response function sends a TakeControl positive response PDU. This function should be called as a response to the **u_takectrl_ind** function being called (a TakeControl indication is received), and after control of the specified semaphore has been successfully granted to the requesting node.

Function Prototype:

```
ST_RET mp_takectrl_resp (MMSREQ_IND *ind,
                         TAKECTRL_RESP_INFO *info);
```

Parameters:

`ind` This pointer to the indication control structure of type **MMSREQ_IND** is received in the **u_takectrl_ind** function.

`info` This pointer to the Operation-Specific data structure of type **TAKECTRL_RESP_INFO** contains information specific to the response PDU to be sent.

Return Value: `ST_RET`

SD_SUCCESS. No Error.

`<> 0` Error Code.

Corresponding User Indication Function: **u_takectrl_ind**

Operation-Specific Data Structure Used: **TAKECTRL_RESP_INFO**

See page 3-13 for a detailed description of this structure.

u_mp_takectrl_conf

Usage: This primitive user confirmation function is called when a confirm to a TakeControl request (**mp_takectrl**) is received. **resp_err** contains a value indicating whether an error occurred. **resp_info_ptr** may contain data that the remote node sent back in the confirmation or it may contain error information if an error occurred. If no error has occurred (**req->resp_err = CNF_RESP_OK**), TakeControl information is available to the application using the **req->resp_info_ptr** pointer. See **Volume 1 — Module 1 — Primitive User Confirmation Function Class** for more information on the recommended handling of confirms.

Note that the function called for this event may be changed if desired; however, **u_mp_takectrl_conf** is the standard default function.

Function Prototype:

```
ST_VOID u_mp_takectrl_conf (MMSREQ_PEND *req);
```

Parameters:

`req` This pointer to the **MMSREQ_PEND** structure is returned from the original **mp_takectrl** request function. See **Volume 1 — Module 2 — Request and Indication Control Data Structures** for more information on this structure.

For a positive response (**req->resp_err = CNF_RESP_OK**), the request control structure member **resp_info_ptr** is a pointer to the operation-specific data structure (**TAKECTRL_RESP_INFO**) for the TakeControl function.

For a negative response, or other errors, see **Module 11 — MMS-EASE Error Handling** starting on page 3-339 for more information.

Return Value: `ST_VOID` (ignored)

Operation-Specific Data Structure Used: **TAKECTRL_RESP_INFO**

See page 3-13 for a detailed description of this structure.

5. RelinquishControl Service

This service is used to release control of a semaphore, previously granted using the TakeControl service. This is accomplished by deleting a specified semaphore entry in the list of owners of the semaphore, and modifying the state of the semaphore.

Note: It is up the user's application at the server to perform these actions. MMS-EASE does not do this.

Primitive Level RelinquishControl Operations

The following section contains information on how to use the paired primitive interface for the Relinquish-Control service. It covers available data structures used by the PPI, and the four primitive level functions that together make up the RelinquishControl Service. See **Volume 1 — Module 2 — General Sequence of Events — Confirmed Services** for a detailed explanation of the sequence of events that occur within the MMS-EASE environment during confirmed services.

- The RelinquishControl service consists of the paired primitive functions of **`mp_relctrl`**, **`u_relctrl_ind`**, **`mp_relctrl_resp`**, and **`u_mp_relctrl_conf`**.

Data Structures

Request/Indication

The operation-specific structure described below is used by the client in issuing a RelinquishControl request (**`mp_relctrl`**). It is received by the server when a RelinquishControl indication (**`u_relctrl_ind`**) is received.

```
struct relctrl_req_info
  {
  OBJECT_NAME sem_name;
  ST_BOOLEAN  named_token_pres;
  ST_CHAR     named_token [MAX_IDENT_LEN+1];
  SD_END_STRUCT
  };
typedef struct relctrl_req_info  RELCTRL_REQ_INFO;
```

Fields:

`sem_name` This structure of type **`OBJECT_NAME`** defines the name of the semaphore for which control is to be relinquished. Semaphore names may only be VMD-specific or domain-specific. See **Volume 1 — Module 2 — MMS Object Name Structure** for more information on this structure.

`named_token_pres` **`SD_FALSE`**. Do Not include **`named_token`** in the PDU.

`SD_TRUE`. Include **`named_token`** in the PDU.

`named_token` This is used only if the semaphore is a pooled semaphore. It should be the same name provided by the responding node when control of the semaphore was granted using the TakeControl service.

Paired Primitive Interface Functions

mp_relctrl

Usage: This primitive request function sends a RelinquishControl request PDU using the data from a structure of type **RELCTRL_REQ_INFO**, pointed to by **info**. This service is used to relinquish control over a semaphore at the remote node for which control was granted using a previous TakeControl service request.

Function Prototype:

```
MMSREQ_PEND *mp_relctrl (ST_INT chan,
                         RELCTRL_REQ_INFO *info);
```

Parameters:

`chan` This integer contains the channel number over which the RelinquishControl PDU is to be sent.

`info` This pointer to the Operation-Specific data structure of type **RELCTRL_REQ_INFO** contains information specific to the request PDU to be sent.

Return Value:

`MMSREQ_PEND *` This is a pointer to the queue data structure of type **MMSREQ_PEND** used to send the PDU. In case of an error, the pointer is set to null and **mms_op_err** is written with the error code.

Corresponding User Confirmation Function: **u_mp_relctrl_conf**

Operation-Specific Data Structure Used: **RELCTRL_REQ_INFO**

See page 3-19 for a detailed description of this structure.

u_relctrl_ind

Usage: This user indication function is called when a RelinquishControl indication is received by a server node. The contents and operation of this function are user-defined, and depend on the application requirements. To conform to the MMS protocol requirements, the application must do one of the following some time after the indication is received:

1) send a positive response using the primitive response function (**`mp_relctrl_resp`**) after successfully releasing the specified semaphore from its controlled state and deleting the corresponding Semaphore Entry, or

2) call the appropriate primitive negative (error) response function (**`mp_err_resp`**). See **Module 11 — MMS-EASE Error Handling** on page 3-343 for an explanation of this function.

See **Volume 1 — Module 1 — User Indication Function Class** for additional information regarding the recommended handling of confirms.

Function Prototype: `ST_VOID u_relctrl_ind (MMSREQ_IND *ind);`

Parameters:

`ind` This pointer to the **`MMSREQ_IND`** structure is described in **Volume 1 — Module 2 — Indication and Request Control Data Structures**. This pointer must be used to send the response to this indication. The primary item of interest in this structure is **`ind->req_info_ptr`**. This is a pointer to the operation-specific data structure for a RelinquishControl indication (**`RELCTRL_REQ_INFO`**). This pointer will always be valid when **`u_relctrl_ind`** is called.

Return Value: `ST_VOID` (ignored)

Operation-Specific Data Structure Used: **`RELCTRL_REQ_INFO`**

See page 3-19 for a detailed description of this structure.

mp_relctrl_resp

Usage: This positive response function sends a RelinquishControl positive response PDU. This function should be called as a response to the **u_relctrl_ind** function being called (received a RelinquishControl indication), and after successfully releasing the specified semaphore from its controlled state, and deleting the corresponding Semaphore Entry.

Function Prototype: `ST_RET mp_relctrl_resp (MMSREQ_IND *ind);`

Parameters:

`ind` This pointer to the indication control structure of type **MMSREQ_IND** is received in the **u_relctrl_ind** function.

Return Value: `ST_RET` **SD_SUCCESS**. No Error.

`<> 0` Error Code.

Corresponding User Indication Function: **u_relctrl_ind**

Operation-Specific Data Structure Used: NONE

u_mp_relctrl_conf

Usage: This primitive user confirmation function is called when a confirm to a RelinquishControl request (**mp_relctrl**) is received. **resp_err** contains a value indicating whether an error occurred. **resp_info_ptr** may contain data that the remote node sent back in the confirmation or it may contain error information if an error occurred. See **Volume 1 — Module 1 — Primitive User Confirmation Function Class** for specific information on the recommended handling of confirms.

Note that the function called for this event may be changed if desired; however, **u_mp_relctrl_conf** is the standard default function.

Function Prototype:

```
ST_VOID u_mp_relctrl_conf (MMSREQ_PEND *req);
```

Parameters:

`req` This pointer of structure type **MMSREQ_PEND** is returned from the original **mp_relctrl** request function. See **Volume 1 — Module 2 — Request and Indication Control Data Structures** for more information on the structure.

For a positive response (**req->resp_err = CNF_RESP_OK**), the request control structure member **resp_info_pres** indicates that there is no confirm data for this function.

For a negative response, or other errors, see **Module 11 — MMS-EASE Error Handling** starting on page 3-339 for more information.

Return Value: `ST_VOID` (ignored)

Operation-Specific Data Structure Used: NONE

6. DefineSemaphore Service

This service creates a Token Semaphore at a client used to coordinate activities with another node.

NOTE: It is up to the user application at the server to create the semaphore queue and state tables, and to provide the code for managing this semaphore. MMS-EASE does not do this.

Primitive Level DefineSemaphore Operations

The following section contains information on how to use the paired primitive interface for the DefineSemaphore service. It covers available data structures used by the PPI, and the four primitive level functions that together make up the DefineSemaphore Service. See **Volume 1 — Module 2 — General Sequence of Events — Confirmed Services** for a detailed explanation of the sequence of events that occur within the MMS-EASE environment during confirmed services.

- The DefineSemaphore service consists of the paired primitive functions of **`mp_defsem`**, **`u_defsem_ind`**, **`mp_defsem_resp`**, and **`u_mp_defsem_conf`**.

Data Structures

Request/Indication

The operation-specific structure described below is used by the client in issuing a DefineSemaphore request (**`mp_defsem`**). It is received by the server when a DefineSemaphore indication (**`u_defsem_ind`**) is received.

```
struct defsem_req_info
  {
  OBJECT_NAME sem_name;
  ST_UINT16   num_of_tokens;
  SD_END_STRUCT
  };
typedef struct defsem_req_info DEFSEM_REQ_INFO;
```

Fields:

`sem_name` This structure of type **`OBJECT_NAME`** defines the name of the semaphore. See **Volume 1 — Module 2 — MMS Object Name Structure** for more information on this structure. Semaphore names may only be VMD-specific or domain-specific.

`num_of_tokens` This indicates the maximum number of owners allowed for this semaphore.

Paired Primitive Interface Functions

mp_defsem

Usage: This primitive request function sends a DefineSemaphore request PDU using the data from a structure of type **DEFSEM_REQ_INFO**, pointed to by **info**. This service is used to define a token semaphore (not a pool semaphore) at a remote node.

Function Prototype:
```
MMSREQ_PEND *mp_defsem (ST_INT chan,
                        DEFSEM_REQ_INFO *info);
```

Parameters:

chan	This integer contains the channel number over which the PDU is to be sent.
info	This pointer to the Operation-Specific data structure of type **DEFSEM_REQ_INFO** contains information specific to the request PDU to be sent.

Return Value:

MMSREQ_PEND *	This pointer to the request control data structure of type **MMSREQ_PEND** is used to send the PDU. In case of an error, the pointer is set to null and **mms_op_err** is written with the error code.

Corresponding User Confirmation Function: **u_mp_defsem_conf**

Operation-Specific Data Structure Used: **DEFSEM_REQ_INFO**

See page 3-25 for a detailed description of this structure.

u_defsem_ind

Usage: This user indication function is called when a DefineSemaphore indication is received by a server node. The contents and operation of this function are user-defined, and depend on the application requirements. To conform to the MMS protocol requirements, the application must do one of the following some time after the indication is received:

1) send a positive response using the primitive response function (**`mp_defsem_resp`**) after the specified semaphore has been successfully defined and created, or

2) call the appropriate primitive negative (error) response function (**`mp_err_resp`**) within the user indication function. See **Module 11 — MMS-EASE Error Handling** on page 3-343 for an explanation of this function.

See **Volume 1 — Module 1 — User Indication Function Class** for additional information on recommended handling of indications.

Function Prototype:

```
ST_VOID u_defsem_ind (MMSREQ_IND *ind);
```

Parameters:

`ind` This pointer to the **`MMSREQ_IND`** structure is described in **Volume 1 — Module 2 — Request and Indication Control Data Structures**. This pointer must be used to send the response to this indication. The primary item of interest in this structure is **`ind->req_info_ptr`**. This is a pointer to the operation-specific data structure for the DefineSemaphore indication (**`DEFSEM_REQ_INFO`**). This pointer will always be valid when **`u_defsem_ind`** is called.

Return Value: `ST_VOID` (ignored)

Operation-Specific Data Structure Used: **`DEFSEM_REQ_INFO`**

See page 3-25 for a detailed description of this structure.

mp_defsem_resp

Usage: This primitive response function sends a DefineSemaphore positive response PDU. This function should be called after the **u_defsem_ind** function is called (a DefineSemaphore indication is received), and after the specified semaphore has been successfully defined and created. Do not allow a semaphore to be defined if:

1. the semaphore already exists;
2. the semaphore cannot be created due to lack of resources;
3. the MMS-user making the request lacks sufficient privilege.

Function Prototype: `ST_RET mp_defsem_resp (MMSREQ_IND *ind);`

Parameters:

`ind` This pointer to the indication control structure of type **MMSREQ_IND** is received in the **u_defsem_ind** function.

Return Value: `ST_RET` **SD_SUCCESS**. No Error

`<> 0` Error Code

Corresponding User Indication Function: **u_defsem_ind**

Operation-Specific Data Structure Used: NONE

u_mp_defsem_conf

Usage: This primitive user confirmation function is called when a confirm to a DefineSemaphore request (**mp_defsem**) is received. **resp_err** contains a value indicating whether an error occurred. **resp_info_ptr** may contain data that the remote node sent back in the confirmation or it may contain error information if an error occurred. **See Volume 1 — Module 1 — Primitive User Confirmation Function Class** for specific information on the recommended handling of confirms.

Note that the function called for this event may be changed if desired; however, **u_mp_defsem_conf** is the standard default function.

Function Prototype:

```
ST_VOID u_mp_defsem_conf (MMSREQ_PEND *req);
```

Parameters:

`req` This pointer to the **MMSREQ_PEND** structure is returned from the original **mp_defsem** request function. See **Volume 1 — Module 2 — Request and Indication Control Data Structures** for information on the structure.

For a positive response (**req->resp_err = CNF_RESP_OK**), the request control structure member **resp_info_pres** indicates that there is no confirm data for this function.

For a negative response, or other errors, see **Module 11 — MMS-EASE Error Handling** starting on page 3-339 for more information.

Return Value: ST_VOID (ignored)

Operation-Specific Data Structure Used: NONE

7. DeleteSemaphore Service

This service is used to delete a semaphore whose attribute is MMS deletable.

Primitive Level DeleteSemaphore Operations

The following section contains information on how to use the paired primitive interface for the DeleteSemaphore service. It covers available data structures used by the PPI, and the four primitive level functions that together make up the DeleteSemaphore Service. See **Volume 1 — Module 2 — General Sequence of Events — Confirmed Services** for a detailed explanation of the sequence of events that occur within the MMS-EASE environment during confirmed services.

- The DeleteSemaphore service consists of the paired primitive functions of **`mp_delsem`**, **`u_delsem_ind`**, **`mp_delsem_resp`**, and **`u_mp_delsem_conf`**.

Data Structures

Request/Indication

The operation-specific structure described below is used by the client in issuing a DeleteSemaphore request (**`mp_delsem`**). It is received by the server when a DeleteSemaphore indication (**`u_delsem_ind`**) is received.

```
struct delsem_req_info
  {
  OBJECT_NAME  sem_name;
  };
typedef struct delsem_req_info DELSEM_REQ_INFO;
```

Fields:

`sem_name` This structure of type **`OBJECT_NAME`** describes the name of the semaphore to be deleted. Semaphore names may only be VMD-specific, or domain-specific. See **Volume 1 — Module 2 — MMS Object Name Structure** for more information on this structure.

Paired Primitive Interface Functions

mp_delsem

Usage: This primitive request function sends a DeleteSemaphore request PDU using the data from a structure of type **DELSEM_REQ_INFO**, pointed to by **info**. This service is used to delete a token semaphore at a remote node. Only MMS deletable token semaphores not currently owned, can be remotely deleted in this manner. This information can be obtained by using the ReportSemaphoreStatus function (**mp_rsstat_resp**) and examining the information in the **RSSTAT_RESP_INFO** structure. See pages 3-41 for a description of the function and 3-38 for a description of the structure for more information. Semaphores created using a DefineSemaphore service request are normally MMS deletable.

Function Prototype:

```
MMSREQ_PEND *mp_delsem (ST_INT chan,
                        DELSEM_REQ_INFO *info);
```

Parameters:

`chan` This integer contains the channel number over which the PDU is to be sent.

`info` This pointer to the Operation-Specific data structure of type **DELSEM_REQ_INFO** contains information specific to the request PDU to be sent.

Return Value:

`MMSREQ_PEND *` This pointer to the request control data structure of type **MMSREQ_PEND** sends the PDU. In case of an error, the pointer is set to null and **mms_op_err** is written with the error code.

Corresponding User Confirmation Function: **u_mp_delsem_conf**

Operation-Specific Data Structure Used: **DELSEM_REQ_INFO**

See page 3-31 for a detailed description of this structure.

u_delsem_ind

Usage: This user indication function is called when a DeleteSemaphore indication is received by a server node. The contents and operation of this function are user-defined, and depend on the application requirements. To conform to the MMS protocol requirements, the application must do one of the following some time after the indication is received:

1) send a positive response using the primitive response function (**`mp_delsem_resp`**) after the specified semaphore has been successfully deleted, or

2) call the appropriate primitive negative (error) response function (**`mp_err_resp`**). See **Module 11 — MMS-EASE Error Handling** on page 3-343 for an explanation of this function.

See **Volume 1 — Module 1 — User Indication Function Class** for additional information on recommended handling of indications.

Function Prototype:

```
ST_VOID u_delsem_ind (MMSREQ_IND *ind);
```

Parameters:

`ind` This pointer to the **`MMSREQ_IND`** structure is described in **Volume 1 — Module 2 — Request and Indication Control Data Structures**. This pointer must be used to send the response to this indication. The primary item of interest in this structure is **`ind->req_info_ptr`**. This is a pointer to the operation-specific data structure for a DeleteSemaphore indication (**`DELSEM_REQ_INFO`**). This pointer will always be valid when **`u_delsem_ind`** is called.

Return Value: `ST_VOID` (ignored)

Operation-Specific Data Structure Used: **`DELSEM_REQ_INFO`**

See page 3-31 for a detailed description of this structure.

mp_delsem_resp

Usage: This primitive response function sends a DeleteSemaphore positive response PDU. This function should be called after the **`u_delsem_ind`** function is called (a DeleteSemaphore indication is received), and after the specified semaphore has been successfully deleted. Do not allow remote deletions of a semaphore if:

1. the semaphores does not exist;
2. it is not MMS deletable;
3. it has at least one active owner; or
4. the remote node lacks sufficient privilege.

Function Prototype: `ST_RET mp_delsem_resp (MMSREQ_IND *ind);`

Parameters:

`ind` This pointer to the indication control structure of type **`MMSREQ_IND`** is received in the **`u_delsem_ind`** function.

Return Value: `ST_RET` **`SD_SUCCESS`**. No Error.

`<> 0` Error Code.

Corresponding User Indication Function: **`u_delsem_ind`**

Operation-Specific Data Structure Used: NONE

u_mp_delsem_conf

Usage: This primitive user confirmation function is called when a confirm to a DeleteSemaphore request (**mp_delsem**) is received. **resp_err** contains a value indicating whether an error occurred. **resp_info_ptr** may contain data that the remote node sent back in the confirmation or it may contain error information if an error occurred. See **Volume 1 — Module 1 — Primitive User Confirmation Function Class** for specific information on the recommended handling of confirms.

Note that the function called for this event may be changed if desired; however, **u_mp_delsem_conf** is the standard default function.

Function Prototype:

```
ST_VOID u_mp_delsem_conf (MMSREQ_PEND *req);
```

Parameters:

`req` This pointer to the **MMSREQ_PEND** structure is returned from the original **mp_delsem** request function. See **Volume 1 — Module 2 — Request and Indication Control Data Structures** for information on the structure.

For a positive response (**req->resp_err = CNF_RESP_OK**), the request control structure member **resp_info_pres** indicates that there is no confirm data for this function.

For a negative response, or other errors, see **Module 11 — MMS-EASE Error Handling** starting on page 3-339 for more information.

Return Value: `ST_VOID` (ignored)

Operation Specific Data Structure Used: NONE

8. ReportSemaphoreStatus Service

This service is used to report to the requesting MMS-user the summarized attributes of a semaphore.

Primitive Level ReportSemaphoreStatus Operations

The following section contains information on how to use the paired primitive interface for the ReportSemaphoreStatus service. It covers available data structures used by the PPI, and the four primitive level functions that together make up the ReportSemaphoreStatus service. See **Volume 1 — Module 2 — General Sequence of Events — Confirmed Services** for a detailed explanation of the sequence of events that occur within the MMS-EASE environment during confirmed services.

- The ReportSemaphoreStatus service consists of the paired primitive functions of **`mp_rsstat`**, **`u_rsstat_ind`**, **`mp_rsstat_resp`**, and **`u_mp_rsstat_conf`**.

Data Structures

Request/Indication

The operation-specific structure described below is used by the client in issuing a ReportSemaphoreStatus (**`mp_rsstat`**). It is received by the server when a ReportSemaphoreStatus indication (**`u_rsstat_ind`**) is received.

```
struct rsstat_req_info
  {
  OBJECT_NAME  sem_name;
  };
typedef struct rsstat_req_info RSSTAT_REQ_INFO;
```

Fields:

`sem_name` This structure of type **`OBJECT_NAME`** defines the name of the semaphore. Semaphore names may only be VMD-specific or domain-specific. See **Volume 1 — Module 2 — MMS Object Name Structure** for an explanation of this structure.

Response/Confirm

The operation-specific data structure described below is used by the server in issuing a ReportSemaphoreStatus response (**mp_rsstat_resp**). It is received by the client when a ReportSemaphoreStatus confirm (**u_mp_rsstat_conf**) is received.

```
struct rsstat_resp_info
  {
  ST_BOOLEAN mms_deletable;
  ST_INT16   tclass;
  ST_UINT16   num_of_tokens;
  ST_UINT16   num_of_owned;
  ST_UINT16   num_of_hung;
  SD_END_STRUCT
  };
typedef struct rsstat_resp_info  RSSTAT_RESP_INFO;
```

Fields:

mms_deletable		**SD_FALSE**. This semaphore is NOT deletable using a MMS service request.
		SD_TRUE. This semaphore is deletable using a DeleteSemaphore service request.
tclass	**0**	The semaphore is a token semaphore
	1	The semaphore is a pool semaphore
num_of_tokens		This indicates the maximum number of owners allowed for this semaphore.
num_of_owned		This indicates the current number of owners of the semaphore (the associated Semaphore Entry is not HUNG).
num_of_hung		This indicates the current number of owned tokens whose Semaphore Entry is HUNG.

Paired Primitive Interface Functions

mp_rsstat

Usage: This primitive request function sends a ReportSemaphoreStatus request PDU using the data from a structure of type **RSSTAT_REQ_INFO**, pointed to by **info**. This service is used to request a summarized status of a semaphore from the remote node.

Function Prototype:

```
MMSREQ_PEND *mp_rsstat (ST_INT chan,
                        RSSTAT_REQ_INFO *info);
```

Parameters:

chan	This integer contains the channel number over which the PDU is to be sent.
info	This pointer to the Operation-Specific data structure of type **RSSTAT_REQ_INFO** contains information specific to the request PDU to be sent.

Return Value:

MMSREQ_PEND *	This pointer to the request control data structure of type **MMSREQ_PEND** sends the PDU. In case of an error, the pointer is set to null and **mms_op_err** is written with the error code.

Corresponding User Confirmation Function: **u_mp_rsstat_conf**

Operation-Specific Data Structure Used: **RSSTAT_REQ_INFO**

See page 3-37 for a detailed description of this structure.

u_rsstat_ind

Usage: This user indication function is called when a ReportSemaphoreStatus indication is received by a server node. The contents and operation of this function are user-defined, and depend on the application requirements. To conform to the MMS protocol requirements, the application must do one of the following some time after the indication is received:

1) send a positive response using the primitive response function (**mp_rsstat_resp**) after the status of the specified semaphore has been successfully obtained, or

2) call the appropriate primitive negative (error) response function (**mp_err_resp**). See **Module 11 — MMS-EASE Error Handling** on page 3-343 for an explanation of this function.

See **Volume 1 — Module 1 — User Indication Function Class** for additional explanation regarding recommended handling of indications.

Function Prototype:

```
ST_VOID u_rsstat_ind (MMSREQ_IND *ind);
```

Parameters:

`ind` This pointer to the **MMSREQ_IND** structure is described in **Volume 1 — Module 2 — Request and Indication Control Data Structures**. This pointer must be used to send the response to this indication. The primary item of interest in this structure is **ind->req_info_ptr**. This is a pointer to the operation-specific data structure for a ReportSemaphoreStatus indication (**RSSTAT_REQ_INFO**). This pointer will always be valid when **u_rsstat_ind** is called.

Return Value: `ST_VOID` (ignored)

Operation-Specific Data Structure Used: **RSSTAT_REQ_INFO**

See page 3-37 for a detailed description of this structure.

mp_rsstat_resp

Usage: This primitive response function sends a ReportSemaphoreStatus positive response PDU. This function should be called after the **u_rsstat_ind** function is called (a ReportSemaphoreStatus indication is received), and after the status of the specified semaphore has been successfully obtained.

Function Prototype:

```
ST_RET mp_rsstat_resp (MMSREQ_IND *ind,
                       RSSTAT_RESP_INFO *info);
```

Parameters:

`ind` This pointer to the indication control structure of type **MMSREQ_IND** is received in the **u_rsstat_ind** function.

`info` This pointer to an operation-specific data structure of type **RSSTAT_RESP_INFO** contains information specific to the response PDU to be sent.

Return Value: `ST_RET` **SD_SUCCESS**. No Error

`<> 0` Error Code

Corresponding User Indication Function: **u_rsstat_ind**

Operation-Specific Data Structure Used: **RSSTAT_RESP_INFO**

See page 3-38 for a detailed description of this structure.

u_mp_rsstat_conf

Usage: This primitive user confirmation function is called when a confirm to a ReportSemaphoreStatus request (**mp_rsstat**) is received. **resp_err** contains a value indicating whether an error occurred. **resp_info_ptr** may contain data that the remote node sent back in the confirmation or it may contain error information if an error occurred. If no error has occurred (**req->resp_err = CNF_RESP_OK**), ReportSemaphoreStatus information is available to the application using the **req->resp_info_ptr** pointer. See **Volume 1 — Module 1 — Primitive User Confirmation Function Class** for specific information on the recommended handling of confirms.

Note that the function called for this event may be changed if desired; however, **u_mp_rsstat_conf** is the standard default function.

Function Prototype:

```
ST_VOID u_mp_rsstat_conf (MMSREQ_PEND *req);
```

Parameters:

`req` This pointer to the **MMSREQ_PEND** structure is returned from the original **mp_rsstat** request function. See **Volume 1 — Module 2 — Indication and Request Control Data Structures** for information on the structure.

For a positive response (**req->resp_err = CNF_RESP_OK**), the request control structure member **resp_info_ptr** is a pointer to the operation-specific data structure (**RSSTAT_RESP_INFO**) for the ReportSemaphoreStatus.

For a negative response, or other errors, see **Module 11 — MMS-EASE Error Handling** starting on page 3-339 for more information.

Return Value: `ST_VOID` (ignored)

Operation-Specific Data Structure Used: **RSSTAT_RESP_INFO**

See page 3-38 for a detailed description of this structure.

9. ReportPoolSemaphoreStatus Service

This service is used to report to the MMS-user the attributes (name and state) of the list of named tokens controlled by a pool semaphore.

Primitive Level ReportPoolSemaphoreStatus Operations

The following section contains information on how to use the paired primitive interface for the ReportPoolSemaphoreStatus service. It covers available data structures used by the PPI, and the four primitive level functions that together make up the ReportPoolSemaphoreStatus service. See **Volume 1 — Module 2 — General Sequence of Events — Confirmed Services** for a detailed explanation of the sequence of events that occur within the MMS-EASE environment during confirmed services.

- The ReportPoolSemaphoreStatus service consists of the paired primitive functions of **`mp_rspool`**, **`u_rspool_ind`**, **`mp_rspool_resp`**, and **`u_mp_rspool_conf`**.

Data Structures

Request/Indication

The operation-specific structure described below is used by the client in issuing a ReportPoolSemaphoreStatus (**`mp_rspool`**). It is received by the server when a ReportPoolSemaphoreStatus indication (**`u_rspool_ind`**) is received.

```
struct rspool_req_info
  {
  OBJECT_NAME sem_name;
  ST_BOOLEAN  start_after_pres;
  ST_CHAR     start_after [MAX_IDENT_LEN+1];
  SD_END_STRUCT
  };
typedef struct rspool_req_info RSPOOL_REQ_INFO;
```

Fields:

`sem_name` This structure of type **`OBJECT_NAME`** defines the name of the pool semaphore for which the status is being requested. Semaphores may only be VMD-specific or domain-specific. See **Volume 1 — Module 2 — MMS Object Name Structure** for an explanation of this structure.

`start_after_pres` **`SD_FALSE`**. Do Not include **`start_after`** in the PDU. Start with the first name in the list.

`SD_TRUE`. Include **`start_after`** in the PDU. This is used during subsequent requests when the entire list of pool semaphore status could not be returned in the response to the original request.

`start_after` This pointer to the data specifies the **`entry_id`** of the Named Token. This indicates the Named Token that the list should begin after if the first name is not desired. The **`entry_id`** is an octet string assigned by the VMD managing the pool semaphore, and used to uniquely identify a given named token from other named tokens of the pool semaphore.

Response/Confirm

The operation-specific data structure described below is used by the server in issuing a ReportPoolSemaphoreStatus response (**mp_rspool_resp**). It is received by the client when a ReportPoolSemaphoreStatus confirm (**u_mp_rspool_conf**) is received.

```
struct rspool_resp_info
  {
  ST_INT16   num_of_tokens;
  ST_BOOLEAN more_follows;
/*TOKEN_ID   named_token_list [num_of_tokens];      */
  SD_END_STRUCT
  };
typedef struct rspool_resp_info  RSPOOL_RESP_INFO;
```

Parameters:

num_of_tokens	This indicates the number of named tokens contained in this response.
more_follows	**SD_TRUE**. There are more named tokens available
	SD_FALSE. This is the end of the named token list.
named_token_list	This array of structures of type **TOKEN_ID** (see below) contains information regarding each named token in this response.

NOTE: FOR RESPONSE ONLY, when allocating a data structure of type **RSPOOL_RESP_INFO**, enough memory must be allocated to hold the information for the array of structures containing the named token list in the **named_token_list[]** member of this structure. The following C statement can be used:

```
info = (RSPOOL_RESP_INFO *) chk_malloc(sizeof (RSPOOL_RESP_INFO) + num_of_tokens
             * sizeof(TOKEN_ID));
```

The following structure contains two versions. The compact form of the structure is used by MMS-EASE *Lite* only.

```
#if !defined (USE_COMPACT_MMS_STRUCTS)

struct token_id
  {
  ST_INT16 token_tag;
  ST_CHAR  named_token[MAX_IDENT_LEN+1];
  SD_END_STRUCT
  };

#else        /* Use compact form */

struct token_id
  {
  ST_INT16  token_tag;
  ST_CHAR    *named_token;
  SD_END_STRUCT
  };
#endif

typedef struct token_id   TOKEN_ID;
```

Fields:

`token_tag` This is a tag indicating the current state of the named token as follows:

- **`0`** The named token is a free named token.
- **`1`** The named token is an owned named token.
- **`2`** The named token is a hung named token.

`named_token` This MMS identifier (a null-terminated character string) identifies the specific named token of this pool semaphore.

Paired Primitive Interface Functions

mp_rspool

Usage: This primitive request function sends a ReportPoolSemaphoreStatus request PDU using the data from a structure of type **RSPOOL_REQ_INFO**, pointed to by **info**. This service is used to request a summarized status of a pool semaphore from the remote node.

Function Prototype:
```
MMSREQ_PEND *mp_rspool (ST_INT chan,
                        RSPOOL_REQ_INFO *info);
```

Parameters:

`chan` This integer contains the channel number over which the PDU is to be sent.

`info` This pointer to the Operation-Specific data structure of type **RSPOOL_REQ_INFO** contains information specific to the request PDU to be sent.

Return Value:

`MMSREQ_PEND *` This pointer to the request control data structure of type **MMSREQ_PEND** is used to send the PDU. In case of an error, the pointer is set to null and **mms_op_err** is written with the error code.

Corresponding User Confirmation Function: **u_mp_rspool_conf**

Operation-Specific Data Structure Used: **RSPOOL_REQ_INFO**

See page 3-43 for a detailed description of this structure.

u_rspool_ind

Usage: This user indication function is called when a ReportPoolSemaphoreStatus indication is received by a server node. The contents and operation of this function are user-defined, and depend on the application requirements. To conform to the MMS protocol requirements, the application must do one of the following some time after the indication is received:

1) send a positive response using the primitive response function (**`mp_rspool_resp`**) after the status of the named token(s) controlled by the specified pool semaphore has been successfully obtained, or

2) call the appropriate primitive negative (error) response function (**`mp_err_resp`**). See **Module 11 — MMS-EASE Error Handling** on page 3-343 for an explanation of this function.

See **Volume 1 — Module 1 — User Indication Function Class** for additional information regarding the recommended handling of indications.

Function Prototype: `ST_VOID u_rspool_ind (MMSREQ_IND *ind);`

Parameters:

`ind` This pointer to the **`MMSREQ_IND`** structure is described in **Volume 1 — Module 2 — Request and Indication Control Data Structures**. This pointer must be used to send the response to this indication. The primary item of interest in this structure is **`ind->req_info_ptr`**. This is a pointer to the operation-specific data structure ReportPoolSemaphoreStatus indication (**`RSPOOL_REQ_INFO`**). This pointer will always be valid when **`u_rspool_ind`** is called.

Return Value: `ST_VOID` (ignored)

Operation-Specific Data Structure Used: **`RSPOOL_REQ_INFO`**

See page 3-43 for a detailed description of this structure.

mp_rspool_resp

Usage: This primitive response function sends a ReportPoolSemaphoreStatus positive response PDU. This function should be called after the **u_rspool_ind** function is called (a ReportPoolSemaphoreStatus indication is received), and after the status of the named token(s) controlled by the specified pool semaphore has been successfully obtained. The status of a pool semaphore cannot be obtained if:

1. The semaphore name is unknown, or cannot be accessed; or
2. the semaphore name does not reference a pool semaphore.

Function Prototype:

```
ST_RET mp_rspool_resp (MMSREQ_IND *ind,
                       RSPOOL_RESP_INFO *info);
```

Parameters:

`ind` This pointer to the indication control structure of type **MMSREQ_IND** is received in the **u_rspool_ind** function.

`info` This pointer to an Operation-Specific data structure of type **RSPOOL_RESP_INFO** contains information specific to the PDU to be sent.

Return Value: `ST_RET` **SD_SUCCESS**. No Error

`<> 0` Error Code

Corresponding User Indication Function: **u_rspool_ind**

Operation-Specific Data Structure Used: **RSPOOL_RESP_INFO**

See page 3-44 for a detailed description of this structure.

u_mp_rspool_conf

Usage: This primitive user confirmation function is called when a confirm to a ReportPoolSemaphoreStatus request (**mp_rspool**) is received. **resp_err** contains a value indicating whether an error occurred. **resp_info_ptr** may contain data that the remote node sent back in the confirmation or it may contain error information if an error occurred. If no error has occurred (**req->resp_err = CNF_RESP_OK**), report pool semaphore status information is available to the application using the **req->resp_info_ptr** pointer. See **Volume 1 — Module 1 — Primitive User Confirmation Function Class** for specific information on the recommended handling of confirms.

Note that the function called for this event may be changed if desired; however, **u_mp_rspool_conf** is the standard default function.

Function Prototype:

```
ST_VOID u_mp_rspool_conf (MMSREQ_PEND *req);
```

Parameters:

`req` This pointer to the **MMSREQ_PEND** structure is returned from the **mp_rspool** request function. See **Volume 1 — Module 2 — Request and Indication Control Data Structures** for more information on this structure.

For a positive response (**req->resp_err = CNF_RESP_OK**), the request control structure member **resp_info_ptr** is a pointer to the operation-specific data structure (**RSPOOL_RESP_INFO**) for the ReportPoolSemaphoreStatus function.

For a negative response, or other errors, see **Module 11 — MMS-EASE Error Handling** starting on page 3-339 for more information.

Return Value: `ST_VOID` (ignored)

Operation-Specific Data Structure Used: **RSPOOL_RESP_INFO**

See page 3-44 for a detailed description of this structure.

10. ReportSemaphoreEntryStatus Service

This service is used to report to the requesting MMS-user the attributes of a list of semaphore entries dependent on a semaphore. This detailed status is sorted by states (QUEUED, OWNER, HUNG).

Primitive Level ReportSemaphoreEntryStatus Operations

The following section contains information on how to use the paired primitive interface for the ReportSemaphoreEntryStatus service. It covers available data structures used by the PPI, and the four primitive level functions that together make up the ReportSemaphoreEntryStatus service. See **Volume 1 — Module 1 — General Sequence of Events — Confirmed Services** for a detailed explanation of the sequence of events that occur within the MMS-EASE environment during confirmed services.

- The ReportSemaphoreEntryStatus service consists of the paired primitive functions of **`mp_rsentry`**, **`u_rsentry_ind`**, **`mp_rsentry_resp`**, and **`u_mp_rsentry_conf`**.

Data Structures

Request/Indication

The operation-specific structure described below is used by the client in issuing a ReportSemaphoreEntryStatus (**`mp_rsentry`**). It is received by the server when a ReportSemaphoreEntryStatus indication (**`u_rsentry_ind`**) is received.

```
struct rsentry_req_info
  {
  OBJECT_NAME sem_name;
  ST_INT16    state;
  ST_BOOLEAN  start_after_pres;
  ST_INT      sa_len;
  ST_UCHAR    *start_after;
  SD_END_STRUCT
  };
typedef struct rsentry_req_info  RSENTRY_REQ_INFO;
```

Fields:

`sem_name` This structure of type **`OBJECT_NAME`** defines the name of the semaphore. Semaphore names may only be VMD-specific or domain-specific. See **Volume 1 — Module 2 — MMS Object Name Structure** for more information on this structure.

`state`

- **`0`** **`QUEUED`**. The request is for the names of the application processes (or the Semaphore Entry) waiting for control of the semaphore.
- **`1`** **`OWNER`**. The request is for the names of the application processes that own the semaphore.
- **`2`** **`HUNG`**. The request is for the names of the application processes that are HUNG possibly because of waiting for control over other semaphores.

`start_after_pres` **`SD_TRUE`**. Include **`start_after`** in the PDU. This is used during subsequent requests when the entire list of semaphore entry status' could not be returned in the response to the original request.

SD_FALSE. Do Not include **start_after** in the PDU. Start with the first name in the list.

sa_len — This contains the length, in bytes, of the data pointed to by **start_after**.

start_after — This pointer to the data specifies the **entry_id** of the Semaphore Entry. This is the name of the Semaphore Entry that the list should begin after if the first name is not desired. The **entry_id** is an octet string assigned by the VMD managing the semaphore entry. It uniquely identifies a given semaphore entry from other semaphore entries of the same semaphore.

Response/Confirm

The operation-specific data structure described below is used by the server in issuing a ReportSemaphoreEntryStatus response (**mp_rsentry_resp**). It is received by the client when a ReportSemaphoreEntryStatus confirm (**u_mp_rsentry_conf**) is received.

```
struct rsentry_resp_info
  {
  ST_INT           num_of_sent;
  ST_BOOLEAN more_follows;
/*SEMAPHORE_ENTRY sent_list [num_of_sent];                        */
  SD_END_STRUCT
  };
typedef struct rsentry_resp_info RSENTRY_RESP_INFO;
```

Fields:

num_of_sent — This indicates the number of semaphore entries contained in this response.

more_follows — **SD_FALSE**. This is the end of semaphore entry list.

SD_TRUE. There are more semaphore entries available.

sent_list — This array of structures of type **semaphore_entry** (see below) contains the information regarding each semaphore entry in this response.

NOTES:

1. See the **TAKECTRL_REQ_INFO** structure on page 3-11 for more information on the meaning of some of the variables of the **SEMAPHORE_ENTRY** structure.
2. FOR RESPONSE ONLY, when allocating a data structure of type **RSENTRY_RESP_INFO**, enough memory must be allocated to hold the **sent_list[]** member of this structure. The following C statement can be used:

```
info = (RSENTRY_RESP_INFO *) chk_malloc(sizeof(RSENTRY_RESP_INFO) + num_of_sent *
            sizeof(SEMAPHORE_ENTRY));
struct semaphore_entry
  {
  ST_INT      ei_len;
  ST_UCHAR    *entry_id;
  ST_CHAR     entry_class;
  ST_INT      app_ref_len;
  ST_UCHAR    *app_ref;
  ST_BOOLEAN named_token_pres;
  ST_CHAR     named_token[MAX_IDENT_LEN+1];
  ST_UCHAR    priority;
  ST_BOOLEAN rem_timeout_pres;
  ST_UINT32  rem_timeout;
  ST_BOOLEAN abrt_on_timeout_pres;
  ST_BOOLEAN abrt_on_timeout;
  ST_BOOLEAN rel_conn_lost;
  SD_END_STRUCT
  };
typedef struct semaphore_entry SEMAPHORE_ENTRY;
```

Fields:

`ei_len` This indicates the length, in bytes, of **`entry_id`**.

`entry_id` This points to the semaphore's entry id. The entry id is an implementation-specific octet string containing data to uniquely identify a semaphore entry from other semaphore entries of the same semaphore.

`entry_class` This indicates the class of the semaphore entry:

- **`0`** **Simple**. The semaphore entry was created using a TakeControl request
- **`1`** **Modifier**. The semaphore entry was created using an AttachToSemaphore Modifier on a MMS service request. See page 3-49 for more information on this modifier.

`app_ref_len` This indicates the length, in bytes, of the data pointed to by **`app_ref`**.

`app_ref` This is a pointer to the AP title of the application process that created this semaphore entry. This field must contain a valid ASN.1 object identifier.

`named_token_pres` **`SD_FALSE`**. Do Not include **`named_token`** in PDU.

`SD_TRUE`. Include **`named_token`** in the PDU.

`named_token` This contains the identifier for the named token if the semaphore entry is for a pool semaphore.

`priority` This contains the priority specified when this semaphore entry was created (using a TakeControl request). If the default value (64) is used, this field is not included in the PDU (highest = 0, lowest = 127, default = 64).

`rem_timeout_pres` **`SD_FALSE`**. Do Not include **`rem_timeout`** in the PDU.

`SD_TRUE`. Include **`rem_timeout`** in the PDU.

`rem_timeout` This contains the remaining acceptable delay, in milliseconds, for either a RelinquishControl or TakeControl timeout if it is in the OWNED or QUEUED state respectively. This should only be present if timeout was specified when the semaphore entry was created.

`abrt_on_timeout_pres` **`SD_FALSE`**. Do Not include **`abrt_on_timeout`** in the PDU.

`SD_TRUE`. Include **`abrt_on_timeout`** in the PDU.

`abrt_on_timeout` This value of **`abrt_on_timeout`** is specified when the semaphore entry was created. It indicates whether to abort the association if the semaphore is owned for more time than that specified by the control timeout parameter.

`rel_conn_lost` This value of **`rel_conn_lost`** is specified when the semaphore entry was created. It indicates whether to relinquish control of the semaphore when the application association that control was granted on is lost.

Paired Primitive Interface Functions

mp_rsentry

Usage: This primitive request function sends a ReportSemaphoreEntryStatus request PDU using the data from a structure of type **RSENTRY_REQ_INFO**, pointed to by **info**. This service is used to obtain the names of the application processes that are either QUEUED, or HUNG for a semaphore, or the OWNER of a semaphore.

Function Prototype:

```
MMSREQ_PEND *mp_rsentry (ST_INT chan,
                         RSENTRY_REQ_INFO *info);
```

Parameters:

chan	This integer contains the channel number over which the PDU is to be sent.
info	This pointer to the Operation-Specific data structure of type **RSENTRY_REQ_INFO** contains information specific to the request PDU to be sent.

Return Value:

MMSREQ_PEND *	This pointer to the request control data structure of type **MMSREQ_PEND** is used to send the PDU. In case of an error, the pointer is set to null and **mms_op_err** is written with the error code.

Corresponding User Confirmation Function: **u_mp_rsentry_conf**

Operation-Specific Data Structure Used: **RSENTRY_REQ_INFO**

See page 3-51 for a detailed description of this structure.

u_rsentry_ind

Usage: This user indication function is called when a ReportSemaphoreEntryStatus indication is received by a server node. The contents and operation of this function are user-defined, and depend on the application requirements. To conform to the MMS protocol requirements, the application must do one of the following some time after the indication is received:

1) send a positive response using the primitive response function (**`mp_rsentry_resp`**) after the specified semaphore entry status information has been successfully obtained, or

2) call the appropriate primitive negative (error) response function (**`mp_err_resp`**). See **Module 11 - MMS-EASE Error Handling** on page 3-343 for an explanation of this function.

See **Volume 1 — Module 1 — User Indication Function Class** for additional information regarding recommended handling of indications.

Function Prototype: `ST_VOID u_rsentry_ind (MMSREQ_IND *ind);`

Parameters:

`ind` This pointer to the **`MMSREQ_IND`** structure is described in **Volume 1 — Module 2 — Request and Indication Control Data Structures**. This pointer must be used to send the response to this indication. The primary item of interest in this structure is **`ind->req_info_ptr`**. This is a pointer to the operation-specific data structure for a ReportSemaphoreEntryStatus indication (**`RSENTRY_REQ_INFO`**). This pointer will always be valid when **`u_rsentry_ind`** is called.

Return Value: `ST_VOID` (ignored)

Operation-Specific Data Structure Used: **`RSENTRY_REQ_INFO`**

See page 3-51 for a detailed description of this structure.

mp_rsentry_resp

Usage: This primitive response function sends a ReportSemaphoreEntryStatus positive response PDU. This should be called after the **u_rsentry_ind** function is called, and after the specified semaphore entry status information has been successfully obtained. This service provides status information regarding the semaphore entries of a semaphore.

Function Prototype:

```
ST_RET mp_rsentry_resp (MMSREQ_IND *ind,
                        RSENTRY_RESP_INFO *info);
```

Parameters:

`ind` This pointer to the indication control structure of type **MMSREQ_IND** is received in the **u_rsentry_ind** function.

`info` This pointer to an Operation-Specific data structure of type **RSENTRY_RESP_INFO** contains information specific to the response PDU to be sent.

Return Value: `ST_RET`

SD_SUCCESS. No Error

`<> 0` Error Code

Corresponding User Indication Function: **u_rsentry_ind**

Operation-Specific Data Structure Used: **RSENTRY_RESP_INFO**

See page 3-52 for a detailed description of this structure.

u_mp_rsentry_conf

Usage: This primitive user confirmation function is called when a confirm to a ReportSemaphoreEntryStatus request (**mp_rsentry**) is received. **resp_err** contains a value indicating whether an error occurred. **resp_info_ptr** may contain data that the remote node sent back in the confirmation or it may contain error information if an error occurred. If no error has occurred (**req->resp_err = CNF_RESP_OK**), report semaphore entry status information is available to the application using the **req->resp_info_ptr** pointer. See **Volume 1 — Module 1 — Primitive User Confirmation Function Class** for specific information on the recommended handling of confirms.

Note that the function called for this event may be changed if desired; however, **u_mp_rsentry_conf** is the standard default function.

Function Prototype:

```
ST_VOID u_mp_rsentry_conf (MMSREQ_PEND *req);
```

Parameters:

`req` This pointer to the **MMSREQ_PEND** structure is returned from the original **mp_rsentry** request function. See **Volume 1 — Module 2 — Request and Indication Control Data Structures** for information on the structure.

For a positive response (**req->resp_err = CNF_RESP_OK**), the request control structure member **resp_info_ptr** is a pointer to the operation-specific data structure (**RSENTRY_RESP_INFO**) for the ReportSemaphoreEntryStatus function.

For a negative response, or other errors, see **Module 11 — MMS-EASE Error Handling** starting on page 3-339 for more information.

Return Value: `ST_VOID` (ignored)

Operation-Specific Data Structure Used: **RSENTRY_RESP_INFO**

See page 3-52 for a detailed description of this structure.

11. Attach To Semaphore Modifier

The **AttachToSemaphore** modifier is provided so that processing can be delayed until control of a semaphore is granted by this service. Services which are modified are not acted on immediately. They are placed in a queue corresponding to a semaphore at the server. When this service rises to the top of the queue, it is acted upon. See **Volume 1 — Module 2** for an explanation of general modifier handling for all confirmed requests.

```
struct attach_to_semaphore
  {
  OBJECT_NAME sem_name;
  ST_BOOLEAN  named_token_pres;
  ST_CHAR      named_token[MAX_IDENT_LEN+1];
  ST_UCHAR    priority;
  ST_BOOLEAN  acc_delay_pres;
  ST_UINT32   acc_delay;
  ST_BOOLEAN  ctrl_timeout_pres;
  ST_UINT32   ctrl_timeout;
  ST_BOOLEAN  abrt_on_timeout_pres;
  ST_BOOLEAN  abrt_on_timeout;
  ST_BOOLEAN  rel_conn_lost;
  SD_END_STRUCT
  };
typedef struct attach_to_semaphore ATTACH_TO_SEMAPHORE;
```

Fields:

`sem_name` This structure of type **`OBJECT_NAME`** contains the name of the semaphore. Semaphore names may only be VMD-specific or domain-specific. See **Volume 1 — Module 2 — MMS Object Name Structure** for a detailed description of this structure.

`named_token_pres` **`SD_FALSE`**. Do Not include **`named_token`** in the PDU.

`SD_TRUE`. Include **`named_token`** in the PDU.

`named_token` This contains the name token of the semaphore if the semaphore is a pool semaphore.

`priority` This specifies the priority that this AttachToSemaphore modifier should have concerning other requests for control of the same semaphore. Use a value of 0 for the highest priority, and 127 for the lowest priority. The default is 64.

`acc_delay_pres` **`SD_FALSE`**. Do Not include **`acc_delay`** in the PDU.

`SD_TRUE`. Include **`acc_delay`** in the PDU.

`acc_delay` This indicates the number of milliseconds of acceptable delay the remote node should wait for control of the semaphore before sending an error response. The default is wait indefinitely.

`ctrl_timeout_pres` **`SD_FALSE`**. Do Not include **`ctrl_timeout`** in the PDU.

`SD_TRUE`. Include **`ctrl_timeout`** in the PDU.

`ctrl_timeout` This indicates the number of milliseconds for which control should be granted. If the amount of time that control is held exceeds this value, the actions taken depend on the values of the **`rel_conn_lost`** and **`abort_on_timeout`** parameters. If not included, control should be granted indefinitely.

`abrt_on_timeout_pres`

`SD_TRUE`. Include **`abrt_on_timeout`** in the PDU.

NOTE: Do not include **`abrt_on_timeout`** field if **`ctrl_timeout`** is not included.

`SD_FALSE`. Do Not include **`abrt_on_timeout`** in the PDU.

NOTE: Must include **`abort_on_timeout`** if including **`ctrl_timeout`**.

`abrt_on_timeout`

`SD_FALSE`. Do not abort the association when the control timeout, **`ctrl_timeout`**, is exceeded.

`SD_TRUE`. Abort the association when the control timeout, **`ctrl_timeout`** is exceeded.

`rel_conn_lost`

`SD_TRUE`. Automatically relinquish control of the semaphore when the association that control was granted over is lost. This includes the case where the association is aborted because of control timeout and **`abort_on_timeout != SD_FALSE`**.

`SD_FALSE`. Do not automatically relinquish control of the semaphore when the association that control was granted over is lost. Instead the semaphore should remain OWNED and the corresponding semaphore entry should be HUNG.

Please refer to page 3-1 for additional information on modifier handling.

12. Event Management Introduction

This portion of MMS-EASE provides services that allow a client to define and manage event objects at a VMD and obtain notifications of event occurrences. In a real sense an event or an alarm is easy to define. For instance, in a process control application, it is common for a control system to generate an alarm when the process variable (e.g., temperature) exceeds a certain preset limit called the high alarm threshold. In a power distribution application, an alarm might be generated when the difference in the phase angle of the current and voltage waveforms of a power line exceeds a certain number of degrees. The MMS Event Management model provides a framework for accessing and managing the network communications aspects of these kinds of events. This is accomplished by defining three named objects that represent:

1. the state of an event (event condition),
2. who to notify about the occurrence of an event (event enrollment), and
3. the action that the VMD should take upon the occurrence of an event (event action).

For many applications, the communication of alarms can be implemented by using MMS services other than the Event Management services. For instance, a simple system can notify a MMS client about the fact that a process variable has exceeded some preset limit by sending the process variable's value to a MMS client using the InformationReport service. Other schemes using other MMS services are also possible. When the application is more complex and requires a more rigorous definition of the event environment in order to ensure interoperability, the MMS event management model should be used.

Event Condition Model

A MMS Event Condition is a named object that represents the current state of some real condition within the VMD. It is important to note that MMS does not define the VMD action (or programming) that causes a change in the state of the event condition. In the process control example given above, an event condition might reflect an IDLE state for when the process variable was not exceeding the value of the high alarm threshold and an ACTIVE state when the process variable did exceed the limit. MMS does not explicitly define the mapping between the high alarm limit and state of the event condition. Even if the high alarm limit is represented by a MMS variable, MMS does not define the necessary configuration or programming needed to create the mapping between the high alarm limit and the state of the event condition. From the MMS point of view, the change in state of the event condition is caused by some autonomous action on the part of the VMD that is not defined by MMS. The event management model defines two classes of event conditions:

- **Network Triggered**. A networked triggered event condition is triggered when a MMS client specifically triggers it using the TriggerEvent service request. Networked triggered events do not have a state (their state is always DISABLED). They are useful for allowing a MMS client to control the execution of event actions and the notifications of event enrollments.
- **Monitored**. A monitored event condition has a state attribute that the VMD sets based upon some local autonomous action. Monitored event conditions can have a Boolean variable associated with them that is used by the VMD to evaluate the state. The VMD periodically evaluates this variable. If the variable is evaluated as TRUE, the VMD sets the event condition state to ACTIVE. When the Boolean variable is evaluated as FALSE, the VMD sets the event condition state to IDLE. Event conditions that are created as a result of a CreateProgramInvocation request with the monitored attribute set to TRUE are monitored event conditions but they do not have an associated Boolean variable.

In addition to the name of the event condition (an object name that reflects its scope), and its class (Networked Triggered or Monitored), MMS defines the following attributes for both network triggered and monitored event conditions:

MMS Deletable This attribute indicates if the event condition can be deleted using a DeleteEventCondition service request.

State This attribute reflects the state of the event condition and can be IDLE, ACTIVE, or DISABLED. Networked triggered events are always DISABLED.

Priority This attribute reflects the relative priority of an event condition object with respect to all other defined event condition objects. Priority is a relative measurement of the VMD's processing priority when it comes to evaluating the state of the event condition as well as the processing of event notification procedures that are invoked when the event condition changes state. A value of zero (0) is the highest priority. A value of 127 is the lowest priority. A value of 64 is the "normal" priority.

Severity This attribute reflects the relative severity of an event condition object with respect to all other defined condition objects. Severity is a relative measure of the affect that a change in the state of the event condition can have on the VMD. A value of zero (0) is the highest severity. A value of 127 is the lowest severity. A value of 64 is the "normal" severity.

Additionally, MMS also defines the following attributes for Monitored event conditions only:

Monitored Variable This is a reference to the underlying Boolean variable whose value the VMD evaluates in determining the state of an event condition. It can be either a named or unnamed variable object. If it is a named object, it must be a variable with the same name (and scope) of the event condition. If the event condition object is locally defined or it was defined using the CreateProgramInvocation request with the monitored attribute set to TRUE, then the value of the monitored variable reference would be equal to UNSPECIFIED. If the monitored variable is deleted, then the value of this reference would be UNDEFINED and the VMD should disable its event notification procedures for this event condition.

Enabled This attribute reflects whether a change in the value of a monitored value (or the state of the associated program invocation if applicable) should cause the VMD to process the event notification procedures for the event condition (TRUE) or not (FALSE). A client can disable an event condition by changing the attribute using the AlterEventConditionMonitoring service request.

Alarm Summary Report This attribute indicates whether (TRUE) or not (FALSE) the event condition should be included in alarm summary reports in response to a GetAlarmSummaryReport service request.

Evaluation Interval This attribute specifies the maximum amount of time, in milliseconds, between successive evaluations of the event condition of the VMD. The VMD may optionally allow clients to change the evaluation interval.

Time of Last Transition to Active
Time of Last Transition to Idle These attributes contain either the time of day or a time sequence number of the last state transitions of the event condition. If the event condition has never been in the IDLE or ACTIVE state, then the value of the corresponding attribute shall be UNDEFINED.

Event Condition Services

The following services are used with Event Conditions.

DefineEventCondition This service is used by a MMS client to create an event condition object. See page 3-69 for more information.

DeleteEventCondition This service is used by a MMS client to delete an event condition object if the MMS deletable attribute is TRUE. See page 3-75 for more information.

GetEventConditionAttributes This service is used by a MMS client to obtain the static attributes of an existing event condition object. See page 3-81 for more information.

ReportEventConditionStatus This service allows a MMS client to obtain the dynamic status of an event condition including its state, the number of event enrollments enrolled in the event condition, whether it is enabled or disabled, and the time of the last transitions to the active and idle states. See page 3-87 for more information.

AlterEventConditionMonitoring This service allows the MMS client to alter the priority, enable or disable the event condition, enable or disable the alarm summary reports, and to change the evaluation interval if the VMD allows the evaluation interval to be changed. See page 3-93 for more information.

GetAlarmSummary This service allows a MMS client to obtain event condition status and attribute information about groups of event conditions. The client can specify several filters for determining which event condition to include in an alarm summary. See page 3-179 for more information.

Event Actions

An event action is a named MMS object that represents the action that the VMD will take when the state of an event condition changes. An event action is optional. When omitted, the VMD would execute its event notification procedures without processing an event action. An event action, when used, is always defined as a confirmed MMS service request. The event action is attached or linked with an event condition when an event enrollment is defined. For example, an event action might be a MMS Read request. If this event action is attached to an event condition (by being referenced in an event enrollment), when the event condition changes state and the event condition is enabled, the VMD would execute this Read service request just as if it had been received by a client. Except that the Read response (either positive or negative) is included in the Event Notification service request that is sent to the MMS client defined for the event enrollment. A confirmed service request must be used (e.g., Start, Read).

Unconfirmed services (e.g., InformationReport, UnsolicitedStatus and EventNotification) and services that must be used in conjunction with other services (e.g., domain upload/download sequences) cannot be used as event actions. In addition to its name, an event action has the following attributes:

MMS Deletable When TRUE, it indicates that the event action can be deleted using a DeleteEventAction service request.

Service Request This attribute is the MMS confirmed service request that the VMD will process when the event condition that the event action is linked with changes state.

Event Action Services

The following services are used with Event Actions.

DefineEventAction This service is used by a MMS client to create an event action object. See page 3-105 for more information.

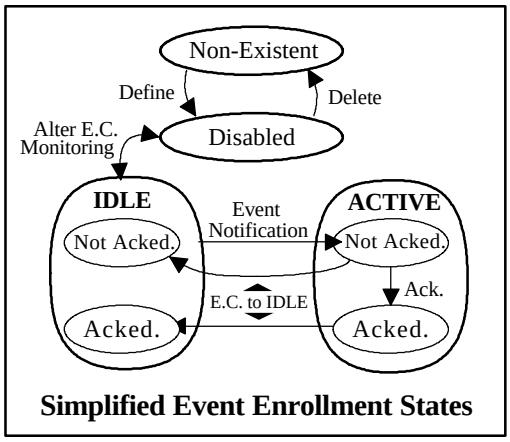
DeleteEventAction This service is used by a MMS client to delete an event action object if the MMS Deletable attribute is TRUE. See page 3-135 for more information.

GetEventActionAttributes This service is used by a MMS client to obtain the attributes of an event action object. See page 3-117 for more information.

ReportEventActionStatus This service allows a MMS client to obtain a list of names of event enrollments that have referenced a given event action. See page 3-123 for more information.

Event Enrollments

An event enrollment is a named MMS object that ties all the elements of the MMS event management model together. The event enrollment represents a request on the part of a MMS client to be notified about changed in the state on an event condition.



Event enrollments have major states (IDLE, ACTIVE) that reflect the event condition (E.C.). Minor states reflect the status of the event acknowledgment process.

Figure 3-3: Simplified Event Enrollment States

When an event enrollment is defined, references are made to an event condition,, an event action (optionally) and the MMS client to which EventNotification should be sent. In addition to its name, the attributes of an event enrollment are:

MMS Deletable If TRUE, this attribute indicates that the event enrollment can be deleted using the DeleteEventEnrollment service request.

Event Condition This attribute is the name of the event condition about which the event enrollment will be notified of changes in state.

Transitions This attribute indicates the state transitions of the event condition for which the VMD should execute its event notification procedures as follows:

- DISABLED-TO-ACTIVE
- DISABLED-TO-IDLE
- IDLE-TO-ACTIVE
- IDLE-TO-DISABLED
- ACTIVE-TO-IDLE
- ACTIVE-TO-DISABLED
- ANY-TO-DELETED

Notification Lost If this attribute is TRUE, it means that the VMD could not successfully complete its event notification procedures due either to 1) some local resource constraint or problem or 2) because the VMD could not establish an association to the client specified in the event enrollment definition. Further transitions of the event condition will be ignored for this event enrollment as long as these problems persist.

Event Action This optional attribute is a reference to the event action that should be processed by the VMD for those state transitions of the event condition specified by the event enrollment's transitions attribute.

Client Application This attribute is a reference to the MMS client to which the EventNotification service requests should be sent for those transitions of the event condition specified by the event enrollment's transitions attribute. This attribute should only be defined if the VMD supports third party services. This attribute is omitted if the duration of the event enrollment is CURRENT.

Duration This attribute indicates the lifetime of the event enrollment. A duration of CURRENT means the event enrollment is only defined for the life of the association between the MMS client and the VMD (similar to an object with application-association specific scope). If the association between the VMD and the client is lost and there is no client application reference attribute for the event enrollment (client application reference is omitted for event enrollments with duration = CURRENT), then the VMD is not capable of re-establishing an application association in order to send an EventNotification to the client. If the duration of the event enrollment is PERMANENT, then the application association between the VMD and the client can be terminated without effecting the event enrollment. In this case, when a specified state transition occurs, the VMD will automatically establish an application association with the specified client.

State The state of an event enrollment indicates IDLE, ACTIVE, DISABLED, and a variety of other states that represent the status of the event notification procedures being executed by the VMD.

Alarm Acknowledgment Rules This attribute specifies the rule of alarm acknowledgment that the VMD should enforce when determining the state of the event enrollment. If an acknowledgment to an EventNotification service request is required, the act of acknowledging (or not acknowledging) the EventNotification shall effect the state of the event enrollment. The various alarm acknowledgment rules are summarized as follows:

- **NONE**. No acknowledgments are required and they shall not effect the state of an event enrollment. These types of event enrollments are not included in alarm enrollment summaries.
- **SIMPLE**. Acknowledgment shall not be required but acknowledgments of transitions to the ACTIVE state shall effect the state of the event enrollment.
- **ACK-ACTIVE**. Acknowledgment of event condition transitions to the ACTIVE state shall be required and shall effect the state of the event enrollment. Acknowledgments of other transitions are optional and shall not effect the state of the event enrollment.
- **ACK-ALL**. Acknowledgments are required for all transitions of the event condition to the ACTIVE or IDLE state and shall effect the state of the event enrollment.

Time Active Acked
Time Idle Acked These attributes reflect the time of the last acknowledgment of the Event Notification for state transitions in the event condition to the ACTIVE or IDLE state corresponding to the event enrollment.

Event Enrollment Services

The following services are used with Event Enrollments:

DefineEventEnrollment This service is used by a MMS client to create an event enrollment object. See page 3-129 for more information.

DeleteEventEnrollment This service is used by a MMS client to delete an event enrollment object is the MMS Deletable attribute is TRUE. See page 3-135 for more information.

GetEventEnrollmentAttributes This service is used by a MMS client to obtain the static attributes of an event enrollment object. See page 3-141 for more information.

ReportEventEnrollmentStatus This service allows the client to obtain the dynamic attributes of an event enrollment including the notification lost, duration, alarm acknowledgment rule and state attributes. See page 3-151 for more information.

AlterEventEnrollment This service allows the client to alter the transitions and alarm acknowledgment rule attributes of an event enrollment. See page 3-159 for more information.

GetAlarmEnrollmentSummary This service allows a MMS client to obtain event enrollment and event condition information about groups of event enrollments. The client can specify several filters for determining which event enrollments to include in an alarm enrollment summary. See page 3-187 for more information.

Event Notification Services

MMS provides services for notifying clients of event condition transitions and acknowledging those event notifications as follows:

EventNotification This unconfirmed service is issued by the VMD to the MMS client to notify the client about event condition transitions that were specified in an event enrollment. There is no response from the client. The acknowledgment of the notification is handled separately. The EventNotification service would include a MMS confirmed service response (positive or negative) if an event action was defined for the event enrollment. See page 3-167 for more information.

AcknowledgeEventNotification This service is used by a MMS client to acknowledge an EventNotification sent to it by the VMD. The client specifies the event enrollment name, the acknowledgment state, and the transition time parameters that were in the EventNotification request that is being acknowledged. See page 3-173 for more information.

TriggerEvent This service is used to trigger a Network Triggered event condition. It gives the client a mechanism by which it can invoke event action and event notification processing by the VMD. For instance, a client can define an event condition, event action, and event enrollments that refer to other MMS clients. When the defining client issues a TriggerEvent service request to the VMD, it will cause the VMD to execute a MMS service request (the event action) and send these results to other MMS clients using the EventNotification service. See page 3-99 for more information.

In addition, there is one service modifier used to operate on the three specified objects, **`attach_to_evcon`**. See page 3-197 for an explanation of handling modifiers.

Common Event Time Data Structure

The following structure is used to hold the time data for the Event Management services of MMS.

```
struct event_time
  {
  ST_INT16      evtime_tag;
  union
    {
    MMS_BTOD    time_of_day;
    ST_UINT32   time_seq_id;
    } evtime;
  };
typedef struct event_time EVENT_TIME;
```

Fields:

`evtime_tag` This is the event time tag:

- **`0`** The **`evtime`** union contains the Binary Time Of Day as defined by the **`MMS_BTOD`** structure. See page 3-5 for more information on this structure.
- **`1`** The **`evtime`** union contains a Time Sequence Identifier as defined by **`time_seq_id.`**
- **`2`** The **`evtime`** union is undefined (null).

`time_of_day` This structure of type **`MMS_BTOD`** contains the Binary Time of Day if **`evtime_tag = 0`**.

`time_seq_id` This is the Time Sequence Identifier if **`evtime_tag = 1`**. A Time Sequence Identifier is an implementation-specific sequence number by which an application can resolve the relative order of events. The relationship between actual time, and the **`time_seq_id`** is determined by the local application.

13. DefineEventCondition Service

This service is used to create an Event Condition object at a VMD.

Primitive Level DefineEventCondition Operations

The following section contains information on how to use the paired primitive interface for the Define EventCondition service. It covers available data structures used by the PPI, and the four primitive level functions that together make up the DefineEventCondition service. See **Volume 1 — Module 2 — General Sequence of Events — Confirmed Services** for a detailed explanation of the sequence of events that occur within the MMS-EASE environment during confirmed services.

- The DefineEventCondition service consists of the paired primitive functions of **`mp_defec`**, **`u_defec_ind`**, **`mp_defec_resp`**, and **`u_mp_defec_conf`**.

Data Structures

Request/Indication

The operation-specific structure described below is used by the client in issuing a DefineEventCondition request (**`mp_defec`**). It is received by the server when a DefineEventCondition indication (**`u_defec_ind`**) is received.

```
struct defec_req_info
  {
  OBJECT_NAME   evcon_name;
  ST_INT16      eclass;
  ST_UCHAR      priority;
  ST_UCHAR      severity;
  ST_BOOLEAN    as_reports_pres;
  ST_BOOLEAN    as_reports;
  ST_BOOLEAN    mon_var_pres;
  VARIABLE_SPEC var_ref;
  ST_BOOLEAN    eval_int_pres;
  ST_UINT32     eval_interval;
  SD_END_STRUCT
  };
typedef struct defec_req_info DEFEC_REQ_INFO;
```

Fields:

`evcon_name` This structure of type **`OBJECT_NAME`** contains the name of the Event Condition to be defined.

`eclass` This indicates the class type of the defined Event Condition:

- **0** network triggered
- **1** monitored (references a variable)

`priority` This indicates the importance of this Event Condition in relation to other Event Condition objects, where 0 = highest, 127 = lowest, and the default = 64.

`severity` This indicates the effect of the Event Condition in relation to other Event Condition objects, where 0 = highest, 255 = lowest, and the default = 255.

`as_reports_pres` **`SD_FALSE`**. Do NOT include **`as_reports`** in the PDU.

`SD_TRUE`. Include **`as_reports`** in the PDU.

`as_reports` **`SD_FALSE`**. Include this Event Condition in alarm summaries **ONLY IF** there is at least one event enrollment with an alarm acknowledgment rule not equal to NONE (**`alarm_ack_rule != SD_FALSE`**). See page 3-129 for more information on this member.

`SD_TRUE`. Include this Event Condition in alarm summaries.

`mon_var_pres` **`SD_FALSE`**. Do NOT include the **`var_ref`** tag in the PDU.

`SD_TRUE`. Include the **`var_ref`** tag in the PDU.

`var_ref` This structure of type **`VARIABLE_SPEC`** contains the specification of the variable monitored by this Event Condition. Use only if **`eclass = 1`**.

`eval_int_pres` **`SD_FALSE`**. Do NOT include **`eval_interval`** in the PDU.

`SD_TRUE`. Include **`eval_interval`** in the PDU.

`eval_interval` This contains the evaluation interval. This is the maximum acceptable time in milliseconds between evaluation of the Event Condition state.

NOTE: See the description in **Volume 1 — Module 2 — MMS Object Name Structure** for more information on **`OBJECT_NAME`** and **Volume 2 — Module 5 — Variable Access** for more information on **`VARIABLE_SPEC`**.

Paired Primitive Interface Functions

mp_defec

Usage: This primitive request function sends a DefineEventCondition request PDU to a remote node. It uses the data found in a structure of type **DEFEC_REQ_INFO**, pointed to by **info**. This service is used to define an Event Condition object.

Function Prototype:

```
MMSREQ_PEND *mp_defec (ST_INT chan,
                       DEFEC_REQ_INFO *info);
```

Parameters:

chan	This is the channel number over which the PDU is to be sent.
info	This pointer to the operation-specific data structure of type **DEFEC_REQ_INFO** contains data specific to the request Condition PDU to be sent.

Return Value:

MMSREQ_PEND *	This is a pointer to the request control data structure of type **MMSREQ_PEND** used to send the PDU. In case of an error, the pointer is set to null and **mms_op_err** is written with the error code.

Corresponding User Confirmation Function: **u_mp_defec_conf**

Operation-Specific Data Structure Used: **DEFEC_REQ_INFO**

See page 3-69 for a detailed description of this structure.

u_defec_ind

Usage: This user indication function is called when a DefineEventCondition indication is received by a server node. The contents and operation of this function are user-defined, and depend on the application requirements. To conform to the MMS protocol requirements, the application must do one of the following some time after the indication is received:

1) send a positive response using the primitive response function (**mp_defec_resp**) after the specified Event Condition has been defined, or

2) call the appropriate primitive negative (error) response function (**mp_err_resp**). See **Module 11 — MMS-EASE Error Handling** on page 3-343 for an explanation of this function.

See **Volume 1 — Module 1 — User Indication Function Class** for additional information regarding the recommended handling of indications.

Function Prototype:

```
ST_VOID u_defec_ind (MMSREQ_IND *ind);
```

Parameters:

`ind` This pointer to the **MMSREQ_IND** structure is described in **Volume 1 — Module 2 — Request and Indication Control Data Structures**. This pointer must be used to send the response to this indication. The primary item of interest in this structure is **ind->req_info_ptr**. This is a pointer to the operation-specific data structure for a DefineEventEnrollment indication (**DEFEC_REQ_INFO**). This pointer will always be valid when **u_defec_ind** is called.

Return Value: `ST_VOID` (ignored)

Operation-Specific Data Structure Used: **DEFEC_REQ_INFO**

See page 3-69 for a detailed description of this structure.

mp_defec_resp

Usage: This primitive response function sends a DefineEventCondition positive response PDU. This function should be called after the **u_defec_ind** function is called (a DefineEventCondition indication is received), and after the specified Event Condition has been defined.

Function Prototype: `ST_RET mp_defec_resp (MMSREQ_IND *ind);`

Parameters:

`ind` This pointer to the indication control structure of type **MMSREQ_IND** is received in the **u_defec_ind** function.

Return Value: `ST_RET` **SD_SUCCESS**. No Error.

`<> 0` Error Code.

Corresponding User Indication Function: **u_defec_ind**

Operation-Specific Data Structure Used: NONE

u_mp_defec_conf

Usage: This primitive user confirmation function is called when a confirm to a DefineEventCondition request (**mp_defec**) is received. **resp_err** contains a value indicating whether an error occurred. **resp_info_ptr** may contain data that the remote node sent back in the confirmation or it may contain error information if an error occurred. If no error has occurred (**req->resp_err = CNF_RESP_OK**), DefineEventCondition information is available to the application using the **req->resp_info_ptr** pointer. See **Volume 1 — Module 1 — Primitive User Confirmation Function Class** for specific information on the recommended handling of confirms.

Note that the function called for this event may be changed if desired; however, **u_mp_defec_conf** is the standard default function.

Function Prototype:

```
ST_VOID u_mp_defec_conf (MMSREQ_PEND *req);
```

Parameters:

`req` This pointer to the **MMSREQ_PEND** structure is returned from the original **mp_defec** request function. See **Volume 1 — Module 2 — Request and Indication Control Data Structures** for more information on this structure.

For a positive response (**req->resp_err = CNF_RESP_OK**), the request control structure member **resp_info_pres** indicates that there is no confirm data for this function.

For a negative response, or other errors, see **Module 11 — MMS-EASE Error Handling** starting on page 3-339 for more information.

Return Value: `ST_VOID` (ignored)

Operation Specific Data Structure Used: NONE

14. DeleteEventCondition Service

This service is used to request that a VMD delete one or more MMS defined Event Condition objects.

Primitive Level DeleteEventCondition Operations

The following section contains information on how to use the paired primitive interface for the DeleteEventCondition service. It covers available data structures used by the PPI, and the four primitive level functions that together make up the DeleteEventCondition service. See **Volume 1 — Module 2 — General Sequence of Events — Confirmed Services** for a detailed explanation of the sequence of events that occur within the MMS-EASE environment during confirmed services.

- The DeleteEventCondition service consists of the paired primitive functions of **`mp_delec`**, **`u_delec_ind`**, **`mp_delec_resp`**, and **`u_mp_delec_conf`**.

Data Structures

Request/Indication

The operation-specific structure described below is used by the client in issuing a DeleteEventCondition request (**`mp_delec`**). It is received by the server when a DeleteEventCondition indication (**`u_delec_ind`**) is received.

```
struct delec_req_info
  {
  ST_INT16    req_tag;
  ST_CHAR     dname [MAX_IDENT_LEN+1];
  ST_INT      num_of_names;
/*OBJECT_NAME name_list [num_of_names];             */
  SD_END_STRUCT
  };
typedef struct delec_req_info DELEC_REQ_INFO;
```

Fields:

`req_tag` This specifies the type of Event Conditions to be deleted:

- **`0`** **SPECIFIC**. Designates the Event Condition objects identified as being the only ones to be deleted.
- **`1`** **AA SPECIFIC**. Designates that all Event Condition objects, whose scope is the current application association, are to be deleted.
- **`2`** **DOMAIN**. Designates that all defined Event Condition objects, whose scope is the specified domain, are to be deleted.
- **`3`** **VMD**. All defined Event Condition objects, whose scope is the server VMD, are to be deleted.

`dname` This is a pointer to the name of the domain from which Event Condition objects will be deleted. This is only used if **`req_tag = 2`**.

`num_of_names` This indicates the number of names of Event Condition objects that are to be deleted. This is only used if **`req_tag = 0`**.

`name_list` This list of structures of type **`OBJECT_NAME`** contains the names of the Event Condition objects to be deleted. See **Volume 1 — Module 2 — MMS Object Name Structure** for an explanation of this structure.

NOTE: FOR REQUEST ONLY, when allocating memory for the **`DELEC_REQ_INFO`** structure, enough memory must be allocated to hold the actual list of Event Condition names contained in **`name_list`**. The following C statement can be used:

```
info = (DELEC_REQ_INFO *) chk_malloc(sizeof(DELEC_REQ_INFO) + (num_of_names *
            sizeof(OBJECT_NAME)));
```

Response/Confirm

The operation-specific data structure described below is used by the server in issuing a DeleteEventCondition response (**`mp_delec_resp`**). It is received by the client when a DeleteEventCondition confirm (**`u_mp_delec_conf`**) is received.

```
struct delec_resp_info
  {
  ST_UINT32   cand_not_deleted;
  };
typedef struct delec_resp_info DELEC_RESP_INFO;
```

Fields:

`cand_not_deleted` This specifies the number of candidates that were not deleted as a result the Delete EventCondition request.

Paired Primitive Interface Functions

mp_delec

Usage: This primitive request function sends a DeleteEventCondition request PDU to a remote node. It uses the data found in a structure of type **DELEC_REQ_INFO**, pointed to by **info**. This service is used to request that a VMD delete one or more MMS defined Event Condition objects.

Function Prototype:

```
MMSREQ_PEND *mp_delec (ST_INT chan,
                       DELEC_REQ_INFO *info);
```

Parameters:

chan	This is the channel number over which the PDU is to be sent.
info	This pointer to the operation-specific data structure, **DELEC_REQ_INFO**, contains data specific to the DeleteEventCondition PDU to be sent.

Return Value:

MMSREQ_PEND *	This pointer to the request control data structure of type **MMSREQ_PEND** is used to send the PDU. In case of an error, the pointer is set to null and **mms_op_err** is written with the error code.

Corresponding User Confirmation Function: **u_mp_delec_conf**

Operation-Specific Data Structure Used: **DELEC_REQ_INFO**

See page 3-75 for a detailed description of this structure.

u_delec_ind

Usage: This user indication function is called when a DeleteEventCondition indication is received by a server node. The contents and operation of this function are user-defined, and depend on the application requirements. To conform to the MMS protocol requirements, the application must do one of the following some time after the indication is received:

1) send a positive response using the primitive response function (**mp_delec_resp**) after the specified Event Condition objects have been deleted, or

2) call the appropriate primitive negative (error) response function (**mp_err_resp**). See **Module 11 — MMS-EASE Error Handling** on page 3-343 for an explanation of this function.

See **Volume 1 — Module 1 — User Indication Function Class** for additional information regarding the recommended handling of indications.

Function Prototype:

```
ST_VOID u_delec_ind  (MMSREQ_IND *ind);
```

Parameters:

`ind` This pointer to the **MMSREQ_IND** structure is described in **Volume 1 — Module 2 — Request and Indication Control Data Structures**. This pointer must be used to send the response to this indication. The primary item of interest in this structure is **ind->req_info_ptr**. This is a pointer to the operation-specific data structure for a DeleteEventCondition indication (**DELEC_REQ_INFO**). This pointer will always be valid when **u_delec_ind** is called.

Return Value: `ST_VOID` (ignored)

Operation-Specific Data Structure Used: **DELEC_REQ_INFO**

See page 3-75 for a detailed description of this structure.

mp_delec_resp

Usage: This primitive response function sends a DeleteEventCondition positive response PDU. This function should be called after the **u_delec_ind** function is called (a DeleteEventCondition indication is received), and after the specified Event Condition objects have been deleted.

Function Prototype:

```
ST_RET mp_delec_resp (MMSREQ_IND *ind,
                      DELEC_RESP_INFO *info);
```

Parameters:

`ind` This pointer to the indication control structure of type **MMSREQ_IND** is received in the **u_delec_ind** function.

`info` This pointer to an Operation-Specific data structure of type **DELEC_RESP_INFO** contains information specific to the response PDU to be sent.

Return Value: `ST_RET` **SD_SUCCESS**. No Error.

`<> 0` Error Code.

Corresponding User Indication Function: **u_delec_ind**

Operation-Specific Data Structure Used: **DELEC_RESP_INFO**

See page 3-76 for a detailed description of this structure.

u_mp_delec_conf

Usage: This primitive user confirmation function is called when a confirm to a **mp_delec** is received. **resp_err** contains a value indicating whether an error occurred. **resp_info_ptr** may contain data that the remote node sent back in the confirmation or it may contain error information if an error occurred. If no error has occurred (**req->resp_err = CNF_RESP_OK**), define event condition information is available to the application using the **req->resp_info_ptr** pointer. See **Volume 1 — Module 1 — Primitive User Confirmation Function Class** for specific information on the recommended handling of confirms.

Note that the function called for this event may be changed if desired; however, **u_mp_delec_conf** is the standard default function.

Function Prototype:

```
ST_VOID u_mp_delec_conf (MMSREQ_PEND *req);
```

Parameters:

`req` This pointer to the **MMSREQ_PEND** structure is returned from the original **mp_delec** request function. See **Volume 1 — Module 2 — Request and Indication Control Data Structures** for more information on this structure.

For a positive response (**req->resp_err = CNF_RESP_OK**), the request control structure member **resp_info_ptr** is a pointer to the operation-specific data structure (**DELEC_RESP_INFO**) for the DeleteEventCondition function.

For a negative response, or other errors, see **Module 11 — MMS-EASE Error Handling** starting on page 3-339 for more information.

Return Value: `ST_VOID` (ignored)

Operation Specific Data Structure Used: **DELEC_RESP_INFO**

See page 3-76 for a detailed description of this structure.

15. GetEventConditionAttributes Service

This service provides the client with the descriptive attributes of an Event Condition object at a VMD.

Primitive Level GetEventConditionAttributes Operations

The following section contains information on how to use the paired primitive interface for the GetEventConditionAttributes service. It covers available data structures used by the PPI, and the four primitive level functions that together make up the GetEventConditionAttributes service. See **Volume 1 — Module 2 — General Sequence of Events — Confirmed Services** for a detailed explanation of the sequence of events that occur within the MMS-EASE environment during confirmed services.

- The GetEventConditionAttributes service consists of the paired primitive functions of **`mp_geteca`**, **`u_geteca_ind`**, **`mp_geteca_resp`**, and **`u_mp_geteca_conf`**.

Data Structures

Request/Indication

The operation-specific structure described below is used by the client in issuing a GetEventConditionAttributes request (**`mp_geteca`**). It is received by the server when a GetEventConditionAttributes indication (**`u_geteca_ind`**) is received.

```
struct geteca_req_info
  {
  OBJECT_NAME  evcon_name;
  };
typedef struct geteca_req_info GETECA_REQ_INFO;
```

Fields:

`evcon_name` This contains the name of the event condition for which the attributes are being requested.

Response/Confirm

The operation-specific data structure described below is used by the server in issuing a GetEventConditionAttributes response (**`mp_geteca_resp`**). It is received by the client when a GetEventConditionAttributes confirm (**`u_mp_geteca_conf`**) is received.

```
struct geteca_resp_info
  {
  ST_BOOLEAN    mms_deletable;
  ST_INT16      eclass;
  ST_UCHAR      priority;
  ST_UCHAR      severity;
  ST_BOOLEAN    as_reports;
  ST_BOOLEAN    mon_var_pres;
  ST_INT16      mon_var_tag;
  VARIABLE_SPEC var_ref;
  ST_BOOLEAN    eval_int_pres;
  ST_UINT32     eval_interval;
  SD_END_STRUCT
  };
typedef struct geteca_resp_info  GETECA_RESP_INFO;
```

Fields:

`mms_deletable` **SD_FALSE**. This event condition is NOT deletable using a service request.

SD_TRUE. This event condition is deletable using a service request.

`eclass` This indicates the class type of the Event Condition:

0 Network triggered

1 Monitored (references a variable)

`priority` This indicates the priority of the Event Condition with 0 = highest, 127 = lowest, 64 = default.

`severity` This indicates the severity of the Event Condition with 0 = most severe, 127 = least severe, 64=default (normal severity).

`as_reports` **SD_TRUE**. Include this Event Condition in alarm summaries.

SD_FALSE. Include this Event Condition in alarm summaries only if there is at least one event enrollment with an alarm acknowledge rule not equal to NONE.

`mon_var_pres` **SD_FALSE**. Do NOT include **mon_var_tag** or **var_ref** in the PDU.

SD_TRUE. Include **mon_var_tag** in the PDU. (Use only if **eclass = 1**).

`mon_var_tag` This is a tag indicating whether there is a monitored defined variable for this event condition:

0 The monitored variable is defined. **var_ref** will be included in the PDU

1 The underlying variable reference is UNDEFINED because the variable has been deleted since the event condition was originally defined. Do not include **var_ref** in the PDU.

`var_ref` This structure of type **VARIABLE_SPEC** contains the variable specification monitored by this event condition. See the section on the variable operation-specific support structures in **Volume 2 — Module 5 — Variable Access**.

`eval_int_pres` **SD_FALSE**. Do NOT include **eval_interval** in the PDU.

SD_TRUE. Include **eval_interval** in the PDU.

`eval_interval` This contains the evaluation interval. This is the maximum acceptable time in milliseconds between evaluations of the Event Conditions state.

Paired Primitive Interface Functions

mp_geteca

Usage: This primitive request function sends a GetEventConditionAttributes request PDU to a remote node. It uses the data found in a structure of type **GETECA_REQ_INFO**, pointed to by **info**. This service is used to obtain a list of the attributes of the specified Event Condition at the remote node.

Function Prototype:

```
MMSREQ_PEND *mp_geteca (ST_INT chan,
                        GETECA_REQ_INFO *info);
```

Parameters:

chan	This is the channel number over which the PDU is to be sent.
info	This pointer to the operation-specific data structure (**GETECA_REQ_INFO**) contains data specific to the GetEventConditionAttributes PDU to be sent.

Return Value:

MMSREQ_PEND *	This pointer to the request control data structure of type **MMSREQ_PEND** is used to send the PDU. In case of an error, the pointer is set to null and **mms_op_err** is written with the error code.

Corresponding User Confirmation Function: **u_mp_geteca_conf**

Operation-Specific Data Structure Used: **GETECA_REQ_INFO**

See page 3-81 for a detailed description of this structure.

u_geteca_ind

Usage: This user indication function is called when a GetEventConditionAttributes indication is received by a server node. The contents and operation of this function are user-defined, and depend on the application requirements. To conform to the MMS protocol requirements, the application must do one of the following some time after the indication is received:

1) send a positive response using the primitive response function (**mp_geteca_resp**) after the specified event condition attributes have been successfully obtained, or

2) call the appropriate primitive negative (error) response function (**mp_err_resp**). See **Module 11 — MMS-EASE Error Handling** on page 3-343 for an explanation of this function.

See **Volume 1 — Module 1 — User Indication Function Class** for additional information regarding the recommended handling of indications.

Function Prototype: `ST_VOID u_geteca_ind (MMSREQ_IND *ind);`

Parameters:

`ind` This pointer to the **MMSREQ_IND** structure is described in **Volume 1 — Module 2 — Request and Indication Control Data Structures**. This pointer must be used to send the response to this indication. The primary item of interest in this structure is **ind->req_info_ptr**. This is a pointer to the operation-specific data structure for the GetEventConditionAttributes indication (**GETECA_REQ_INFO**). This pointer will always be valid when **u_geteca_ind** is called.

Return Value: `ST_VOID` (ignored)

Operation-Specific Data Structure Used: **GETECA_REQ_INFO**

See page 3-81 for a detailed description of this structure.

mp_geteca_resp

Usage: This primitive response function sends a GetEventConditionAttributes positive response PDU. It uses the data from a structure of type **GETECA_RESP_INFO**, pointed to by **info**. This function should be called as a response to the **u_geteca_ind** function being called (a GetEventConditionAttributes indication is received), and after the specified event condition attributes have been successfully obtained.

Function Prototype:

```
ST_RET mp_geteca_resp (MMSREQ_IND *ind,
                       GETECA_RESP_INFO *info);
```

Parameters:

`ind` This pointer to the indication queue data structure of type **MMSREQ_IND** is passed to the **u_geteca_ind** function when it was called.

`info` This pointer to an Operation-Specific data structure of type **GETECA_RESP_INFO** contains information specific to the response PDU to be sent.

Return Value: `ST_RET` **SD_SUCCESS**. No Error.

`<> 0` Error Code.

Corresponding User Indication Function: **u_geteca_ind**

Operation-Specific Data Structure Used: **GETECA_RESP_INFO**

See page 3-81 for a detailed description of this structure.

u_mp_geteca_conf

Usage: This primitive user confirmation function is called when a confirm to a GetEventCondition-Attributes request (**mp_geteca**) is received. **resp_err** contains a value indicating whether an error occurred. **resp_info_ptr** may contain data that the remote node sent back in the confirmation or it may contain error information if an error occurred. If no error has occurred (**req->resp_err = CNF_RESP_OK**), get event condition attributes information is available to the application using the **req->resp_info_ptr** pointer. See **Volume 1 — Module 1 — Primitive User Confirmation Function Class** for specific information on the recommended handling of confirms.

Note that the function called for this event may be changed if desired; however, **u_mp_geteca_conf** is the standard default function.

Function Prototype:

```
ST_VOID u_mp_geteca_conf (MMSREQ_PEND *req);
```

Parameters:

`req` This pointer to the **MMSREQ_PEND** structure is returned from the original **mp_geteca** request function. See **Volume 1 — Module 2 — Request and Indication Control Data Structures** for information on the structure.

For a positive response (**req->resp_err = CNF_RESP_OK**), the request control structure member **resp_info_ptr** is a pointer to the operation-specific data structure (**GETECA_RESP_INFO**) for the GetEventConditionAttributes.

For a negative response, or other errors, see **Module 11 — MMS-EASE Error Handling** starting on page 3-339 for more information.

Return Value: `ST_VOID` (ignored)

Operation-Specific Data Structure Used: **GETECA_RESP_INFO**

See page 3-81 for a detailed description of this structure.

16. ReportEventConditionStatus Service

This service provides the client with the status of an Event Condition object from a VMD.

Primitive Level ReportEventConditionStatus Operations

The following section contains information on how to use the paired primitive interface fro the ReportEventConditionStatus service. It covers available data structures used by the PPI, and the four primitive level functions that together make up the ReportEventConditionStatus service. See **Volume 1 — Module 2 — General Sequence of Events — Confirmed Services** for a detailed explanation of the sequence of events that occur within the MMS-EASE environment during confirmed services.

- The ReportEventConditionStatus service consists of the paired primitive functions of **`mp_repecs`**, **`u_repecs_ind`**, **`mp_repecs_resp`**, and **`u_mp_repecs_conf`**.

Data Structures

Request/Indication

The operation-specific structure described below is used by the client in issuing a ReportEventConditionStatus request (**`mp_repecs`**). It is received by the server when a ReportEventConditionStatus indication (**`u_repecs_ind`**) is received.

```
struct repecs_req_info
  {
  struct OBJECT_NAME  evcon_name;
  };
typedef struct repecs_req_info REPECS_REQ_INFO;
```

Parameters:

`evcon_name` This structure of type **`OBJECT_NAME`** contains the name of the Event Condition for which the status is to be obtained. See **Volume 1 — Module 2 — MMS Object Name Structure** for an explanation of this structure.

Response/Confirm

The operation-specific data structure described below is used by the server in issuing a ReportEventConditionStatus response (**`mp_repecs_resp`**). It is received by the client when a ReportEventConditionStatus confirm (**`u_mp_repecs_conf`**) is received.

```
struct repecs_resp_info
  {
  ST_INT16   cur_state;
  ST_UINT32  num_of_ev_enroll;
  ST_BOOLEAN enabled_pres;
  ST_BOOLEAN enabled;
  ST_BOOLEAN tta_time_pres;
  EVENT_TIME tta_time;
  ST_BOOLEAN tti_time_pres;
  EVENT_TIME tti_time;
  SD_END_STRUCT
  };
typedef struct repecs_resp_info  REPECS_RESP_INFO;
```

Fields:

`curr_state` This indicates the current state of the event condition:

- **0** DISABLED
- **1** IDLE
- **2** ACTIVE

`num_of_ev_enroll` This indicates the number of event enrollments for this event condition. Each event enrollment represents an entity (application process, or Attach to Event Condition modifier) that wants to be informed about transitions of this event condition.

`enabled_pres` **SD_FALSE**. Do NOT include **enabled** in the PDU.

SD_TRUE. Include **enabled** in the PDU. Use only if the event condition is NOT network triggered.

`enabled` **SD_FALSE**. Do NOT process the event enrollments that occur when the monitored variable corresponding to the changes in this event condition.

SD_TRUE. Do process the event enrollments that occur when the monitored variable corresponding to the changes in this event condition.

Note: This field is only used if the event condition is NOT network triggered. See page 3-68 for more information.

`tta_time_pres` **SD_FALSE**. Do NOT include **tta_time** in the PDU.

SD_TRUE. Include **tta_time** in the PDU. Use only if the event condition is NOT network triggered.

`tta_time` This structure of type **EVENT_TIME** contains the time of the last transition to active for this event condition*.

`tti_time_pres` **SD_FALSE**. Do NOT include **tti_time** in the PDU.

SD_TRUE. Include **tti_time** in the PDU. Use only if the event condition is NOT network triggered.

`tti_time` This structure of type **EVENT_TIME** contains the time of the last transition to idle for this event condition[6]. See page 3-53 for an explanation of the **event_time** structure.

[6] * Include these fields only if there has been a corresponding transition **and** the event condition's alarm summary reports attribute is TRUE **or** if the event condition is referenced by an event enrollment whose alarm acknowledgment rule is not NONE.

Paired Primitive Interface Functions

mp_repecs

Usage: This primitive request function sends a ReportEventConditionStatus request PDU to a remote node. It uses the data found in a structure of type **REPECS_REQ_INFO**, pointed to by **info**. This service is used to obtain the status of an Event Condition from a remote node.

Function Prototype:

```
MMSREQ_PEND *mp_repecs (ST_INT chan,
                        REPECS_REQ_INFO *info);
```

Parameters:

chan	This is the channel number over which the PDU is to be sent.
info	This pointer to an operation-specific data structure (**REPECS_REQ_INFO**) contains data specific to the ReportEventCondition Status PDU to be sent.

Return Value:

MMSREQ_PEND *	This pointer to the request control data structure of type **MMSREQ_PEND** sends the PDU. In case of an error, the pointer is set to null and **mms_op_err** is written with the error code.

Corresponding User Confirmation Function: **u_mp_repecs_conf**

Operation-Specific Data Structure Used: **REPECS_REQ_INFO**

See page 3-87 for more information on this structure.

u_repecs_ind

Usage: This user indication function is called when a ReportEventConditionStatus indication is received by a server node. The contents and operation of this function are user-defined, and depend on the application requirements. To conform to the MMS protocol requirements, the application must do one of the following some time after the indication is received:

1) send a positive response using the primitive response function (**mp_repecs_resp**) after the specified Event Condition status information have been successfully obtained, or

2) call the appropriate primitive negative (error) response function (**mp_err_resp**). See **Module 11 — MMS-EASE Error Handling** on page 3-343 for an explanation of this function.

See **Volume 1 — Module 1 — User Indication Function Class** for additional information regarding recommended handling of indications.

Function Prototype:

```
ST_VOID u_repecs_ind (MMSREQ_IND *ind);
```

Parameters:

`ind` This pointer to the **MMSREQ_IND** structure is described in **Volume 1 — Module 2 — Request and Indication Control Data Structures**. This pointer must be used to send the response to this indication. The primary item of interest in this structure is **ind->req_info_ptr**. This is a pointer to the operation-specific data structure for the ReportEventConditionStatus indication (**REPECS_REQ_INFO**). This pointer will always be valid when **u_repecs_ind** is called.

Return Value: `ST_VOID` (ignored)

Operation-Specific Data Structure Used: **REPECS_REQ_INFO**

See page 3-87 for a detailed description of this structure.

mp_repecs_resp

Usage: This primitive response function sends a ReportEventConditionStatus positive response PDU. It uses the data from a structure of type **REPECS_RESP_INFO**, pointed to by **info**. This function should be called as a response to the **u_repecs_ind** function being called (a ReportEventConditionStatus indication is received), and after the specified ReportEventConditionStatus information have been successfully altered.

Function Prototype:

```
ST_RET mp_repecs_resp (MMSREQ_IND *ind,
                       REPECS_RESP_INFO *info);
```

Parameters:

ind	This pointer to the indication control structure of type **MMSREQ_IND** is received in the **u_repecs_ind** function.
info	This pointer to an operation-specific data structure (**REPECS_RESP_INFO**) contains data specific to the ReportEventConditionStatus PDU to be sent.

Return Value:

ST_RET	**SD_SUCCESS**.	No Error.
	<> 0	Error Code.

Corresponding User Indication Function: **u_repecs_ind**

Operation-Specific Data Structure Used: **REPECS_RESP_INFO**

See page 3-87 for a detailed description of this structure.

u_mp_repecs_conf

Usage: This primitive user confirmation function is called when a confirm to a ReportEventConditionStatus request (**mp_repecs**) is received. **resp_err** contains a value indicating whether an error occurred. **resp_info_ptr** may contain data that the remote node sent back in the confirmation or it may contain error information if an error occurred. See **Volume 1 — Module 1 — Primitive User Confirmation Function Class** for specific information on the recommended handling of confirms.

Note that the function called for this event may be changed if desired; however, **u_mp_repecs_conf** is the standard default function.

Function Prototype:

```
ST_VOID u_mp_repecs_conf (MMSREQ_PEND *req);
```

Parameters:

`req` This pointer to the **MMSREQ_PEND** structure is returned from the original **mp_repecs** request. See **Volume 1 — Module 2 — Request and Indication Control Data Structures** for information on the structure.

For a positive response (**req->resp_err = CNF_RESP_OK**), the request control structure member **resp_info_pres** indicates that there is no confirm data for this function.

For a negative response, or other errors, see **Module 11 — MMS-EASE Error Handling** starting on page 3-339 for more information.

Return Value: `ST_VOID` (ignored)

Operation-Specific Data Structure Used: **REPECS_RESP_INFO**

See page 87 for a detailed description of this structure.

17. AlterEventConditionMonitoring Service

This service is used to request that the VMD alter any combination of a monitored Event Condition object's attributes of Enable, Priority, and Alarm Summary Report.

Primitive Level AlterEventConditionMonitoring Operations

The following section contains information on how to use the paired primitive interface for the AlterEventConditionMonitoring service. It covers available data structures used by the PPI, and the four primitive level functions that together make up the AlterEventConditionMonitoring service. See **Volume 1 — Module 2 — General Sequence of Events — Confirmed Services** for a detailed explanation of the sequence of events that occur within the MMS-EASE environment during confirmed services.

- The AlterEventConditionMonitoring service consists of the paired primitive functions of **`mp_altecm`**, **`u_altecm_ind`**, **`mp_altecm_resp`**, and **`u_mp_altecm_conf`**.

Data Structures

Request/Indication

The operation-specific structure described below is used by the client in issuing an AlterEventConditionMonitoring request (**`mp_altecm`**). It is received by the server when an AlterEventConditionMonitoring indication (**`u_altecm_ind`**) is received.

```
struct altecm_req_info
  {
  OBJECT_NAME evcon_name;
  ST_BOOLEAN  enabled_pres;
  ST_BOOLEAN  enabled;
  ST_BOOLEAN  priority_pres;
  ST_UCHAR    priority;
  ST_BOOLEAN  as_reports_pres;
  ST_BOOLEAN  as_reports;
  ST_BOOLEAN  eval_int_pres;
  ST_UINT32   eval_int;
  SD_END_STRUCT
  };
typedef struct altecm_req_info ALTECM_REQ_INFO;
```

Fields:

`evcon_name` This structure of type **`OBJECT_NAME`** contains the name of the Event Condition for which the status is to be obtained. See **Volume 1 — Module 2 — MMS Object Name Structure** for an explanation of this structure.

`enabled_pres` **`SD_FALSE`**. Do NOT include **`enabled`** in the PDU.

`SD_TRUE`. Include **`enabled`** in the PDU.

`enabled` **SD_FALSE**. Do NOT process the Event Enrollments when the monitored variable corresponding to changes in this Event Condition. Use only if the Event Condition **eclass** is NOT network triggered. See page 3-67 for information on the **eclass** attribute.

SD_TRUE. Process these Event Enrollments when the monitored variable corresponding to the changes in this Event Condition. Use only if the Event Condition **eclass** is NOT network triggered.

`priority_pres` **SD_FALSE**. Do NOT include **priority** in the PDU.

SD_TRUE. Include **priority** in the PDU.

`priority` This indicates the priority of the Event Condition where 0 = the highest, 127 = the lowest, and 64 = the default.

`as_report_pres` **SD_FALSE**. Do NOT include **as_reports** in the PDU.

SD_TRUE. Include **as_reports** in the PDU.

`as_reports` **SD_FALSE**. Include this Event Condition in alarm summaries only if there is at least one Event Enrollment with an alarm acknowledgment rule NOT equal to NONE. See page 3-129 for information on the **alarm_ack_rule** attribute.

SD_TRUE. Include this Event Condition in alarm summaries.

`eval_int_pres` **SD_FALSE**. Do NOT include **eval_int** in the PDU.

SD_TRUE. Include **eval_int** in the PDU.

`eval_int` This indicates the maximum amount of time, in milliseconds, between successive evaluations of the event condition by the VMD.

Paired Primitive Interface Functions

mp_altecm

Usage: This primitive request function sends an AlterEventConditionMonitoring request PDU to a remote node. It uses the data found in a structure of type **ALTECM_REQ_INFO**, pointed to by **info**. This service is used to alter any combination of certain attributes associated with an Event Condition such as enabled, priority, and alarm summary reports.

Function Prototype:

```
MMSREQ_PEND *mp_altecm (ST_INT chan,
                        ALTECM_REQ_INFO *info);
```

Parameters:

chan	This is the channel number over which the PDU is to be sent.
info	This pointer to the operation-specific data structure (**ALTECM_REQ_INFO**) contains data specific to the AlterEventConditionMonitoring PDU to be sent.

Return Value:

MMSREQ_PEND *	This pointer to the request control data structure of type **MMSREQ_PEND** sends the PDU. In case of an error, the pointer is set to null and **mms_op_err** is written with the error code.

Corresponding User Confirmation Function: **u_mp_altecm_conf**

Operation-Specific Data Structure Used: **ALTECM_REQ_INFO**

See page 3-93 for a detailed description of this structure.

u_altecm_ind

Usage: This user indication function is called when an AlterEventConditionMonitoring indication is received by a server node. The contents and operation of this function are user-defined, and depend on the application requirements. To conform to the MMS protocol requirements, the application must do one of the following some time after the indication is received:

1) send a positive response using the primitive response function (**mp_altecm_resp**) after the specified Event Condition monitor attributes have been successfully altered, or

2) call the appropriate primitive negative (error) response function (**mp_err_resp**). See **Module 11 — MMS-EASE Error Handling** on page 3-343 for an explanation of this function.

See **Volume 1 — Module 1 — User Indication Function Class** for additional information regarding recommended handling of indications.

Function Prototype: `ST_VOID u_altecm_ind (MMSREQ_IND *ind);`

Parameters:

`ind` This pointer to the **MMSREQ_IND** structure is described in **Volume 1 — Module 2 — Request and Indication Control Data Structures**. This pointer must be used to send the response to this indication. The primary item of interest in this structure is **ind->req_info_ptr**. This is a pointer to the operation-specific data structure for the AlterEventConditionMonitoring indication (**ALTECM_REQ_INFO**). This pointer will always be valid when **u_altecm_ind** is called.

Return Value: `ST_VOID` (ignored)

Operation-Specific Data Structure Used: **ALTECM_REQ_INFO**

See page 3-93 for a detailed description of this structure.

mp_altecm_resp

Usage: This primitive response function sends an AlterEventConditionMonitoring positive response PDU. This function should be called as a response to the **u_altecm_ind** function being called (an AlterEventConditionMonitoring indication is received), and after the specified Event Condition monitor attributes have been successfully altered.

Function Prototype:

```
ST_RET mp_altecm_resp (MMSREQ_IND *ind);
```

Parameters:

`ind` This pointer to the indication control structure of type **MMSREQ_IND** is received in the **u_altecm_ind** function.

Return Value: `ST_RET` **SD_SUCCESS**. No Error.

`<> 0` Error Code.

Corresponding User Indication Function: **u_altecm_ind**

Operation-Specific Data Structure Used: NONE

u_mp_altecm_conf

Usage: This primitive user confirmation function is called when a confirm to an AlterEventConditionMonitoring request (**mp_altecm**) is received. **resp_err** contains a value indicating whether an error occurred. **resp_info_ptr** may contain data that the remote node sent back in the confirmation or it may contain error information if an error occurred. See **Volume 1 — Module 1 — Primitive User Confirmation Function Class** for specific information on the recommended handling of confirms.

Note that the function called for this event may be changed if desired; however, **u_mp_altecm_conf** is the standard default function.

Function Prototype:

```
ST_VOID u_mp_altecm_conf (MMSREQ_PEND *req);
```

Parameters:

`req` This pointer to the **MMSREQ_PEND** structure is returned from the original **mp_altecm** request. See **Volume 1 — Module 2 — Request and Indication Control Data Structures** for information on the structure.

For a positive response (**req->resp_err = CNF_RESP_OK**), the request control structure member **resp_info_pres** indicates that there is no confirm data for this function.

For a negative response, or other errors, see **Module 11 — MMS-EASE Error Handling** starting on page 3-339 for more information.

Return Value: `ST_VOID` (ignored)

Operation-Specific Data Structure Used: NONE

18. TriggerEvent Service

This service is used to request that a VMD trigger an event associated with a network-triggered Event Condition object.

Primitive Level TriggerEvent Operations

The following section contains information on how to use the paired primitive interface for the TriggerEvent service. It covers available data structures used by the PPI, and the four primitive level functions that together make up the TriggerEvent service. See **Volume 1 — Module 2 — General Sequence of Events — Confirmed Services** for a detailed explanation of the sequence of events that occur within the MMS-EASE environment during confirmed services.

- The TriggerEvent service consists of the paired primitive functions of **`mp_trige`**, **`u_trige_ind`**, **`mp_trige_resp`**, and **`u_mp_trige_conf`**.

Data Structures

Request/Indication

The operation-specific structure described below is used by the client in issuing a TriggerEvent request (**`mp_trige`**). It is received by the server when a TriggerEvent indication (**`u_trige_ind`**) is received.

```
struct trige_req_info
  {
  OBJECT_NAME evcon_name;
  ST_BOOLEAN  priority_pres;
  ST_UCHAR    priority;
  SD_END_STRUCT
  };
typedef struct trige_req_info TRIGE_REQ_INFO;
```

Fields:

`evcon_name` This structure of type **`OBJECT_NAME`** contains the name of the Event Condition that is to be triggered. See **Volume 1 — Module 2 — MMS Object Name Structure** for an explanation of this structure.

`priority_pres` **`SD_FALSE`**. Do NOT include **`priority`** in the PDU.

`SD_TRUE`. Include **`priority`** in the PDU.

`priority` This indicates the priority of the Event Condition where 0 = the highest, 127 = the lowest, and 64 = the default.

Paired Primitive Interface Functions

mp_trige

Usage: This primitive request function sends a TriggerEvent request PDU to a remote node. It uses the data found in a structure of type **TRIGE_REQ_INFO**, pointed to by **info**. This service is used to trigger an event associated with a network-triggered Event Condition object.

Function Prototype:

```
MMSREQ_PEND *mp_trige (ST_INT chan,
                       TRIGE_REQ_INFO *info);
```

Parameters:

chan	This is the channel number over which the PDU is to be sent.
info	This pointer to the operation-specific data structure of type **TRIGE_REQ_INFO** contains data specific to the request PDU to be sent.

Return Value:

MMSREQ_PEND *	This pointer to the request control data structure of type **MMSREQ_PEND** is used to send the PDU. In case of an error, the pointer is set to null and **mms_op_err** is written with the error code.

Corresponding User Confirmation Function: **u_mp_trige_conf**

Operation-Specific Data Structure Used: **TRIGE_REQ_INFO**

See page 3-99 for a detailed description of this structure.

u_trige_ind

Usage: This user indication function is called when a TriggerEvent indication is received by a server node. The contents and operation of this function are user-defined, and depend on the application requirements. To conform to the MMS protocol requirements, the application must do one of the following some time after the indication is received:

1) send a positive response using the primitive response function (**`mp_trige_resp`**) after the specified event has been network-triggered with an Event Condition object, or

2) call the appropriate primitive negative (error) response function (**`mp_err_resp`**). See **Module 11 — MMS-EASE Error Handling** on page 3-343 for an explanation of this function.

See **Volume 1 — Module 1 — User Indication Function Class** for additional information regarding recommended handling of indications.

Function Prototype:

```
ST_VOID u_trige_ind  (MMSREQ_IND *ind);
```

Parameters:

`ind` This pointer to the **`MMSREQ_IND`** structure is described in **Volume 1 — Module 2 — Request and Indication Control Data Structures**. This pointer must be used to send the response to this indication. The primary item of interest in this structure is **`ind->req_info_ptr`**. This is a pointer to the operation-specific data structure for a TriggerEvent indication (**`TRIGE_REQ_INFO`**). This pointer will always be valid when **`u_trige_ind`** is called.

Return Value: `ST_VOID` (ignored)

Operation-Specific Data Structure Used: **`TRIGE_REQ_INFO`**

See page 3-99 for a detailed description of this structure.

mp_trige_resp

Usage: This primitive response function sends a TriggerEvent positive response PDU. It should be called as a response to the **u_trige_ind** function being called (a TriggerEvent indication is received), and after the specified Event Condition object has been network-triggered.

Function Prototype: `ST_RET mp_trige_resp (MMSREQ_IND *info);`

Parameters:

`info` This pointer to the indication control structure of type **MMSREQ_IND** is received in the **u_trige_ind** function.

Return Value: `ST_RET` **SD_SUCCESS**. No Error

`<> 0` Error Code

Corresponding User Indication Function: **u_trige_ind**

Operation-Specific Data Structure Used: NONE

u_mp_trige_conf

Usage: This primitive user confirmation function is called when a confirm to a TriggerEvent request (**mp_trige**) is received. **resp_err** contains a value indicating whether an error occurred. **resp_info_ptr** may contain data that the remote node sent back in the confirmation or it may contain error information if an error occurred. See **Volume 1 — Module 1 — Primitive User Confirmation Function Class** for specific information on the recommended handling of confirms.

Note that the function called for this event may be changed if desired; however, **u_mp_trige_conf** is the standard default function.

Function Prototype:

```
ST_VOID u_mp_trige_conf (MMSREQ_PEND *req);
```

Parameters:

`req` This pointer to the **MMSREQ_PEND** structure is returned from the original **mp_trige** request. See **Volume 1 — Module 2 — Request and Indication Control Data Structures** for information on the structure.

For a positive response (**req->resp_err = CNF_RESP_OK**), the request control structure member **resp_info_pres** indicates that there is no confirm data for this function.

For a negative response, or other errors, see **Module 11 — MMS-EASE Error Handling** starting on page 3-339 for more information.

Return Value: `ST_VOID` (ignored)

Operation-Specific Data Structure Used: NONE

19. DefineEventAction Service

This service provides a mechanism so that a client can request the creation of an Event Action object at a VMD.

Primitive Level DefineEventAction Operations

The following section contains information on how to use the paired primitive interface for the DefineEventAction service. It covers available data structures used by the PPI, and the four primitive level functions that together make up the DefineEventAction service. See **Volume 1 — Module 2 — General Sequence of Events — Confirmed Services** for a detailed explanation of the sequence of events that occur within the MMS-EASE environment during confirmed services.

- The DefineEventAction service consists of the paired primitive functions of **mp_defea**, **u_defea_ind**, **mp_defea_resp**, and **u_mp_defea_conf**.

Data Structures

Request/Indication

The operation-specific structure described below is used by the client in issuing a DefineEventAction request (**mp_defea**). It is received by the server when a DefineEventAction indication (**u_defea_ind**) is received.

```
struct defea_req_info
  {
  OBJECT_NAME evact_name;
  ST_INT      conf_serv_req_len;
  ST_UCHAR    *conf_serv_req;
  ST_BOOLEAN  modlist_pres;
  ST_INT       num_of_modifiers;
  ST_BOOLEAN  cs_rdetail_pres;
  ST_INT       cs_rdetail_len;
  ST_UCHAR    *cs_rdetail;
/*MODIFIER    mod_list [num_of_modifiers];                */
  SD_END_STRUCT
  };
typedef struct defea_req_info DEFEA_REQ_INFO;
```

Parameters:

`evact_name` This structure of type **OBJECT_NAME** contains the name of the Event Action to be created. See **Volume 1 — Module 2 — MMS Object Name Structure** for an explanation of this structure.

`conf_serv_req_len` This indicates the length, in bytes, of the data pointed to by **conf_serv_req**.

`conf_serv_req` This is a pointer to a valid argument, specified by the argument parameter of the request and indication service primitives.

`modlist_pres` **SD_FALSE**. Do NOT include **num_of_modifiers** and **mod_list** in the PDU.

SD_TRUE. Include **num_of_modifiers** and **mod_list** in the PDU.

`num_of_modifiers` This contains the number of modifiers in this DefineEventAction PDU.

`cs_rdetail_pres` **SD_FALSE**. Do NOT include **cs_rdetail** in the PDU.

SD_TRUE. Include **cs_rdetail** in the PDU.

`cs_rdetail_len` This indicates the length, in bytes, of the data pointed to by **`cs_rdetail`**.

`cs_rdetail` This is a pointer to the Companion Standard request detail.

`mod_list` This array of structures of type **`MODIFIER`** contains the modifiers present in this modifier list. See page 3-1 for a detailed description of the **`modifier`** structure.

NOTE: When allocating memory for the **`DEFEA_REQ_INFO`** structure, enough memory must be allocated to hold the actual list of modifiers contained in **`mod_list`**. The following C statement can be used:

```
info = (DEFEA_REQ_INFO *) chk_malloc(sizeof (DEFEA_REQ_INFO) + (num_of_modifiers
            * sizeof(MODIFIER)));
```

Paired Primitive Interface Functions

mp_defea

Usage: This primitive request function sends a DefineEventAction request PDU to a remote node. It uses the data found in a structure of type **DEFEA_REQ_INFO**, pointed to by **info**. This service is used to request that an event action object be created.

Function Prototype:

```
MMSREQ_PEND *mp_defea (ST_INT chan,
                       DEFEA_REQ_INFO *info);
```

Parameters:

`chan` This is the channel number over which the PDU is to be sent.

`info` This pointer to the operation-specific data structure of type **DEFEA_REQ_INFO** contains data specific to the request PDU to be sent.

Return Value:

`MMSREQ_PEND *` This pointer to the request control data structure of type **MMSREQ_PEND** is used to send the PDU. In case of an error, the pointer is set to null and **mms_op_err** is written with the error code.

Corresponding User Confirmation Function: **u_mp_defea_conf**

Operation-Specific Data Structure Used: **DEFEA_REQ_INFO**

See page 3-105 for a detailed description of this structure.

u_defea_ind

Usage: This user indication function is called when a DefineEventAction indication is received by a server node. The contents and operation of this function are user-defined, and depend on the application requirements. To conform to the MMS protocol requirements, the application must do one of the following some time after the indication is received:

1) send a positive response using the primitive response function (**mp_defea_resp**) after the specified Event Action object has been created, or

2) call the appropriate primitive negative (error) response function (**mp_err_resp**). See **Module 11 — MMS-EASE Error Handling** on page 3-343 for an explanation of this function.

See **Volume 1 — Module 1 — User Indication Function Class** for additional information regarding recommended handling of indications.

Function Prototype: `ST_VOID u_defea_ind (MMSREQ_IND *ind);`

Parameters:

`ind` This pointer to the **MMSREQ_IND** structure is described in **Volume 1 — Module 2 — Request and Indication Control Data Structures**. This pointer must be used to send the response to this indication. The primary item of interest in this structure is **ind->req_info_ptr**. This is a pointer to the operation-specific data structure for the DefineEventAction indication (**DEFEA_REQ_INFO**). This pointer will always be valid when **u_defea_ind** is called.

Return Value: `ST_VOID` (ignored)

Operation-Specific Data Structure Used: **DEFEA_REQ_INFO**

See page 3-105 for a detailed description of this structure.

mp_defea_resp

Usage: This primitive response function sends a DefineEventAction positive response PDU. This function should be called as a response to the **u_defea_ind** function being called (a DefineEventAction indication is received), and after the specified Event Action object has been created.

Function Prototype: `ST_RET mp_defea_resp (MMSREQ_IND *ind);`

Parameters:

`ind` This pointer to the indication control structure of type **MMSREQ_IND** is received in the **u_defea_ind** function.

Return Value: `ST_RET` **SD_SUCCESS**. No Error.

`<> 0` Error Code.

Corresponding User Indication Function: **u_defea_ind**

Operation-Specific Data Structure Used: NONE

u_mp_defea_conf

Usage: This primitive user confirmation function is called when a confirm to a DefineEventAction request (**mp_defea**) is received. **resp_err** contains a value indicating whether an error occurred. **resp_info_ptr** may contain data that the remote node sent back in the confirmation or it may contain error information if an error occurred. See **Volume 1 — Module 1 — Primitive User Confirmation Function Class** for specific information on the recommended handling of confirms.

Note that the function called for this event may be changed if desired; however, **u_mp_defea_conf** is the standard default function.

Function Prototype:

```
ST_VOID u_mp_defea_conf  (MMSREQ_PEND *req);
```

Parameters:

`req` This pointer to the **MMSREQ_PEND** structure is returned from the original **mp_defea** request. See **Volume 1 — Module 2 — Request and Indication Control Data Structure** for information on the structure.

For a positive response (**req->resp_err = CNF_RESP_OK**), the request control structure member **resp_info_pres** indicates that there is no confirm data for this function.

For a negative response, or other errors, see **Module 11 — MMS-EASE Error Handling** starting on page 3-339 for more information.

Return Value: `ST_VOID` (ignored)

Operation-Specific Data Structure Used: NONE

20. DeleteEventAction Service

This service is used to request that a VMD delete one or more Event Action objects.

Primitive Level DeleteEventAction Operations

The following section contains information on how to use the paired primitive interface for the DeleteEventAction service. It covers available data structures used by the PPI, and the four primitive level functions that together make up the DeleteEventAction service. See **Volume 1 — Module 2 — General Sequence of Events — Confirmed Services** for a detailed explanation of the sequence of events that occur within the MMS-EASE environment during confirmed services.

- The DeleteEventAction service consists of the paired primitive functions of **`mp_delea`, `u_delea_ind`**, **`mp_delea_resp`**, and **`u_mp_delea_conf`**.

Data Structures

Request/Indication

The operation-specific structure described below is used by the client in issuing a DeleteEventAction request (**`mp_delea`**). It is received by the server when a DeleteEventAction indication (**`u_delea_ind`**) is received.

```
struct delea_req_info
  {
  ST_INT16    req_tag;
  ST_CHAR     dname [MAX_IDENT_LEN+1];
  ST_INT      num_of_names;
/*OBJECT_NAME name_list [num_of_names];             */
  SD_END_STRUCT
  };
typedef struct delea_req_info DELEA_REQ_INFO;
```

Fields:

`req_tag` This specifies the type of Event Actions to be deleted:

- **`0`** **SPECIFIC**. Designates the Event Action objects identified as being the only ones to be deleted.
- **`1`** **AA SPECIFIC**. Designates that all Event Action objects, whose scope is the current application association, are to be deleted.
- **`2`** **DOMAIN**. Designates that all defined Event Action objects, whose scope is the specified domain, are to be deleted.
- **`3`** **VMD**. All defined Event Action objects, whose scope is the server VMD, are to be deleted.

`dname` This is a pointer to the name of the domain from which Event Action objects will be deleted. This is only used if **`req_tag = 2`**.

`num_of_names` This indicates the number of names of Event Action objects that are to be deleted. This is only used if **`req_tag = 0`**.

`name_list` This list of structures of type **`OBJECT_NAME`** contains the names of the Event Action objects to be deleted. Used only if **`req_tag = 0`**. See **Volume 1 — Module 2 — MMS Object Name Structure** for an explanation of this structure.

NOTE: FOR REQUEST ONLY, when allocating memory for the **DELEA_REQ_INFO** structure, enough memory must be allocated to hold the actual list of Event Action names contained in **name_list**. The following C statement can be used:

```
info = (DELEA_REQ_INFO *) chk_malloc(sizeof (DELEA_REQ_INFO) + (num_of_names *
            sizeof (OBJECT_NAME)));
```

Response/Confirm

The operation-specific data structure described below is used by the server in issuing a DeleteEventAction response (**mp_delea_resp**). It is received by the client when a DeleteEventAction confirm (**u_mp_delea_conf**) is received.

```
struct delea_resp_info
  {
  ST_UINT32   cand_not_deleted;
  };
typedef struct delea_resp_info DELEA_RESP_INFO;
```

Fields:

`cand_not_deleted` This specifies the number of candidates that were not deleted as a result the Delete EventAction request.

Paired Primitive Interface Functions

mp_delea

Usage: This primitive request function sends a DeleteEventAction request PDU to a remote node. It uses the data found in a structure of type **DELEA_REQ_INFO**, pointed to by **info**. This service is used to delete one or more Event Action objects.

Function Prototype:

```
MMSREQ_PEND *mp_delea (ST_INT chan,
                       DELEA_REQ_INFO *info);
```

Parameters:

chan	This is the channel number over which the PDU is to be sent.
info	This pointer to the operation-specific data structure of type **DELEA_REQ_INFO** contains data specific to the request PDU to be sent.

Return Value:

MMSREQ_PEND *	This pointer to the request control data structure of type **MMSREQ_PEND** is used to send the PDU. In case of an error, the pointer is returned **= 0** and **mms_op_err** is written with the error code.

Corresponding User Confirmation Function: **u_mp_delea_conf**

Operation-Specific Data Structure Used: **DELEA_REQ_INFO**

See page 3-111 for a detailed description of this structure.

u_delea_ind

Usage: This user indication function is called when a DeleteEventAction indication is received by a server node. The contents and operation of this function are user-defined, and depend on the application requirements. To conform to the MMS protocol requirements, the application must do one of the following some time after the indication is received:

1) send a positive response by using the primitive response function (**mp_delea_resp**) after the specified Event Action objects have been deleted, or

2) call the appropriate primitive negative (error) response function (**mp_err_resp**). See **Module 11 — MMS-EASE Error Handling** on page 3-343 for an explanation of this function.

See **Volume 1 — Module 1 — User Indication Function Class** for additional information regarding recommended handling of indications.

Function Prototype:

```
ST_VOID u_delea_ind  (MMSREQ_IND *ind);
```

Parameters:

`ind` This pointer to the **MMSREQ_IND** structure is described in **Volume 1 — Module 2 — Request and Indication Control Data Structures**. This pointer must be used to send the response to this indication. The primary item of interest in this structure is **ind->req_info_ptr**. This is a pointer to the operation-specific data structure for a DeleteEventAction indication (**DELEA_REQ_INFO**). This pointer will always be valid when **u_delea_ind** is called.

Return Value: `ST_VOID` (ignored)

Operation-Specific Data Structure Used: **DELEA_REQ_INFO**

See page 3-111 for a detailed description of this structure.

mp_delea_resp

Usage: This primitive response function sends a DeleteEventAction positive response PDU using the data from a structure of type **DELEA_RESP_INFO**, pointed to by **info**. This function should be called as a response to the **u_delea_ind** function being called (a DeleteEventAction indication is received), and after the specified Event Action objects have been deleted.

Function Prototype:

```
ST_RET mp_delea_resp (MMSREQ_IND *ind,        DELEA_RESP_INFO
                                  *info);
```

Parameters:

`ind` This pointer to the indication control structure of type **MMSREQ_IND** is received in the **u_delea_ind** function.

`info` This pointer to an Operation Specific data structure of type **DELEA_RESP_INFO** contains information specific to the PDU to be sent.

Return Value: `ST_RET` **SD_SUCCESS**. No Error

`<> 0` Error Code

Corresponding User Indication Function: **u_delea_ind**

Operation-Specific Data Structure Used: **DELEA_RESP_INFO**

See page 3-112 for a detailed description of this structure.

u_mp_delea_conf

Usage: This primitive user confirmation function is called when a confirm to a DeleteEventAction request (**mp_delea**) is received. **resp_err** contains a value indicating whether an error occurred. **resp_info_ptr** may contain data that the remote node sent back in the confirmation or it may contain error information if an error occurred. If no error has occurred (**req->resp_err = CNF_RESP_OK**), define event action information is available to the application using the **req->resp_info_ptr** pointer. See **Volume 1 — Module 1 — Primitive User Confirmation Function Class** for specific information on the recommended handling of confirms.

Note that the function called for this event may be changed if desired; however, **u_mp_delea_conf** is the standard default function.

Function Prototype:

```
ST_VOID u_mp_delea_conf (MMSREQ_PEND *req);
```

Parameters:

`req` This pointer to the **MMSREQ_PEND** structure is returned from the original **mp_delea** request. See **Volume 1 — Module 2 — Request and Indication Control Data Structures** for information on the structure.

For a positive response (**req->resp_err = CNF_RESP_OK**), the request control structure member **resp_info_ptr** is a pointer to the operation-specific data structure (**DELEA_RESP_INFO**) for the DeleteEventAction function.

For a negative response, or other errors, see **Module 11 — MMS-EASE Error Handling** starting on page 3-339 for more information.

Return Value: `ST_VOID` (ignored)

Operation-Specific Data Structure Used: **DELEA_RESP_INFO**

See page 3-112 for a detailed description of this structure.

21. GetEventActionAttributes Service

This service provides a mechanism so that a client obtains from a VMD the attributes of an Event Action object.

Primitive Level GetEventActionAttributes Operations

The following section contains information on how to use the paired primitive interface for the GetEventActionAttributes service. It covers available data structures used by the PPI, and the four primitive level functions that together make up the GetEventActionAttributes service. See **Volume 1 — Module 2 — General Sequence of Events — Confirmed Services** for a detailed explanation of the sequence of events that occur within the MMS-EASE environment during confirmed services.

- The GetEventActionAttributes service consists of the paired primitive functions of **`mp_geteaa`**, **`u_geteaa_ind`**, **`mp_geteaa_resp`**, and **`u_mp_geteaa_conf`**.

Data Structures

Request/Indication

The operation-specific structure described below is used by the client in issuing a GetEventActionAttributes request (**`mp_geteaa`**). It is received by the server when a GetEventActionAttributes indication (**`u_geteaa_ind`**) is received.

```
struct geteaa_req_info
  {
  OBJECT_NAME evact_name;
  };
typedef struct geteaa_req_info GETEAA_REQ_INFO;
```

Fields:

`evact_name` This structure of type **`OBJECT_NAME`** contains the name of the Event Action whose attributes are to be obtained. See **Volume 1 — Module 2 — MMS Object Name Structure** for an explanation of this structure.

Response/Confirm

The operation-specific data structure described below is used by the server in issuing a GetEventActionAttributes response (**`mp_geteaa_resp`**). It is received by the client when a GetEventActionAttributes confirm (**`u_mp_geteaa_conf`**) is received.

```
struct geteaa_resp_info
  {
  ST_BOOLEAN  mms_deletable;
  ST_INT      conf_serv_req_len;
  ST_UCHAR    *conf_serv_req;
  ST_BOOLEAN  cs_rdetail_pres;
  ST_INT      cs_rdetail_len;
  ST_UCHAR    *cs_rdetail;
  ST_INT      num_of_modifiers;
/*MODIFIER    mod_list [num_of_modifiers];                */
  SD_END_STRUCT
  };
typedef struct geteaa_resp_info  GETEAA_RESP_INFO;
```

Fields:

`mms_deletable`	**`SD_FALSE`**. Event Action is NOT deletable using the DeleteEventAction service request.
	`SD_TRUE`. Event Action is deletable using the DeleteEventAction service request.
`conf_serv_req_len`	This indicates the length, in bytes, of the data pointed to by **`conf_serv_req`**.
`conf_serv_req`	This is a pointer to a valid argument, specified by the Argument parameter of the request and indication service primitives.
`cs_rdetail_pres`	**`SD_FALSE`**. Do NOT include **`cs_rdetail`** in the PDU.
	`SD_TRUE`. Include **`cs_rdetail`** in the PDU.
`cs_rdetail_len`	This indicates the length, in bytes, of the data pointed to by **`cs_rdetail`**.
`cs_rdetail`	This is a pointer to the Companion Standard request detail.
`num_of_modifiers`	This indicates the number of modifiers in this GetEventActionAttributes PDU.
`mod_list`	This array of structures of type **`MODIFIER`** contains the modifiers present in this modifier list.

NOTE: When allocating memory for the **`GETEAA_REQ_INFO`** structure, enough memory must be allocated to hold the actual list of modifiers contained in **`mod_list`**. The following C statement can be used:

```
info = (GETEAA_REQ_INFO *) chk_malloc(sizeof (GETEAA_REQ_INFO) +
             (num_of_modifiers * sizeof(MODIFIER)));
```

See page 3-1 for a detailed description of the **`MODIFIER`** structure.

Paired Primitive Interface Functions

mp_geteaa

Usage: This primitive request function sends a GetEventActionAttributes request PDU to a remote node. It uses the data found in a structure of type **GETEAA_REQ_INFO**, pointed to by **info**. This service is used to obtain the attributes of an Event Action object.

Function Prototype:

```
MMSREQ_PEND *mp_geteaa (ST_INT chan,
                        GETEAA_REQ_INFO *info);
```

Parameters:

chan	This is the channel number over which the PDU is to be sent.
info	This pointer to the operation-specific data structure of type **GETEAA_REQ_INFO** contains data specific to the request PDU to be sent.

Return Value:

MMSREQ_PEND *	This pointer to the request control data structure of type **MMSREQ_PEND** is used to send the PDU. In case of an error, the pointer is set to null and **mms_op_err** is written with the error code.

Corresponding User Confirmation Function: **u_mp_geteaa_conf**

Operation-Specific Data Structure Used: **GETEAA_REQ_INFO**

See page 3-117 for a detailed description of this structure.

u_geteaa_ind

Usage: This user indication function is called when a GetEventActionAttributes indication is received by a server node. The contents and operation of this function are user-defined, and depend on the application requirements. To conform to the MMS protocol requirements, the application must do one of the following some time after the indication is received:

1) send a positive response using the primitive response function (**`mp_geteaa_resp`**) after the attributes have been obtained from the specified Event Action object, or

2) call the appropriate primitive negative (error) response function (**`mp_err_resp`**). See **Module 11 — MMS-EASE Error Handling** on page 3-343 for an explanation of this function.

See **Volume 1 — Module 1 — User Indication Function Class** for additional information regarding recommended handling of indications.

Function Prototype: `ST_VOID u_geteaa_ind (MMSREQ_IND *ind);`

Parameters:

`ind` This pointer to the **`MMSREQ_IND`** structure is described in **Volume 1 — Module 2 — Request and Indication Control Data Structures**. This pointer must be used to send the response to this indication. The primary item of interest in this structure is **`ind->req_info_ptr`**. This is a pointer to the operation-specific data structure for the GetEventActionAttributes indication (**`GETEAA_REQ_INFO`**). This pointer will always be valid when **`u_geteaa_ind`** is called.

Return Value: `ST_VOID` (ignored)

Operation-Specific Data Structure Used: **`GETEAA_REQ_INFO`**

See page 3-117 for a detailed description of this structure.

mp_geteaa_resp

Usage: This primitive response function sends a GetEventActionAttributes positive response PDU. It uses the data from a structure of type **GETEAA_RESP_INFO**, pointed to by **info**. This function should be called as a response to the **u_geteaa_ind** function being called (a GetEventActionAttributes indication is received), and after the attributes have been obtained from the specified Event Action object.

Function Prototype:

```
ST_RET mp_geteaa_resp (MMSREQ_IND *ind,
                       GETEAA_RESP_INFO *info);
```

Parameters:

`ind` This pointer to the indication control structure of type **MMSREQ_IND** is received in the **u_geteaa_ind** function.

`info` This pointer to an Operation Specific data structure of type **GETEAA_RESP_INFO** contains information specific to the PDU to be sent.

Return Value: `ST_RET` **SD_SUCCESS**. No Error.

`<> 0` Error Code.

Corresponding User Indication Function: **u_geteaa_ind**

Operation-Specific Data Structure Used: **GETEAA_RESP_INFO**

See page 3-117 for a detailed description of this structure.

u_mp_geteaa_conf

Usage: This primitive user confirmation function is called when a confirm to a GetEventActionAttributes request (**mp_geteaa**) is received. **resp_err** contains a value indicating whether an error occurred. **resp_info_ptr** may contain data that the remote node sent back in the confirmation or it may contain error information if an error occurred. If no error has occurred (**req->resp_err = CNF_RESP_OK**), GetEventActionAttributes information is available to the application using the **req->resp_info_ptr** pointer. See **Volume 1 — Module 1 — Primitive User Confirmation Function Class** for specific information on the recommended handling of confirms.

Note that the function called for this event may be changed if desired; however, **u_mp_geteaa_conf** is the standard default function.

Function Prototype:

```
ST_VOID u_mp_geteaa_conf (MMSREQ_PEND *req);
```

Parameters:

`req` This pointer to the **MMSREQ_PEND** structure is returned from the original **mp_geteaa** request. See **Volume 1 — Module 2 — Request and Indication Control Data Structures** for information on the structure.

For a positive response (**req->resp_err = CNF_RESP_OK**), the request control structure member **resp_info_ptr** is a pointer to the operation-specific data structure (**GETEAA_RESP_INFO**) for the GetEventActionAttributes function.

For a negative response, or other errors, see **Module 11 — MMS-EASE Error Handling** starting on page 3-339 for more information.

Return Value: `ST_VOID` (ignored)

Operation-Specific Data Structure Used: **GETEAA_RESP_INFO**

See page 3-117 for a detailed description of this structure.

22. ReportEventActionStatus Service

This service is used to obtain a count of the number of Event Enrollment objects currently specifying an Event Action object.

Primitive Level ReportEventActionStatus Operations

The following section contains information on how to use the paired primitive interface for the ReportEventActionStatus service. It covers available data structures used by the PPI, and the four primitive level functions that together make up the ReportEventActionStatus service. See **Volume 1 — Module 2 — General Sequence of Events — Confirmed Services** for a detailed explanation of the sequence of events that occur within the MMS-EASE environment during confirmed services.

- The ReportEventActionStatus service consists of the paired primitive functions of **`mp_repeas`**, **`u_repeas_ind`**, **`mp_repeas_resp`**, and **`u_mp_repeas_conf`**.

Data Structures

Request/Indication

The operation-specific structure described below is used by the client in issuing a ReportEventActionStatus request (**`mp_repeas`**). It is received by the server when a ReportEventActionStatus indication (**`u_repeas_ind`**) is received.

```
struct repeas_req_info
  {
  OBJECT_NAME  evact_name;
  };
typedef struct repeas_req_info REPEAS_REQ_INFO;
```

Fields:

`evact_name` This structure of type **`OBJECT_NAME`** contains the name of the Event Action for which a count of associated Event Enrollment objects is to be obtained. See **Volume 1 — Module 2 — MMS Object Name Structure** for an explanation of this structure.

Response/Confirm

The operation-specific data structure described below is used by the server in issuing a ReportEventActionStatus response (**`mp_repeas_resp`**). It is received by the client when a ReportEventActionStatus confirm (**`u_mp_repeas_conf`**) is received.

```
struct repeas_resp_info
  {
  ST_UINT32   num_of_ev_enroll;
  };
typedef struct repeas_resp_info  REPEAS_RESP_INFO;
```

Fields:

`num_of_ev_enroll` This indicates the number of Event Enrollments for this Event Action. Each Event Enrollment represents an entity (application process, or AttachToEventCondition modifier) that wants to be informed about transitions of the Event Action.

Paired Primitive Interface Functions

mp_repeas

Usage: This primitive request function sends a ReportEventActionStatus request PDU to a remote node. It uses the data found in a structure of type **REPEAS_REQ_INFO**, pointed to by **info**. This service is used to obtain a count of the Event Enrollments currently specifying an Event Action object.

Function Prototype:

```
MMSREQ_PEND *mp_repeas (ST_INT chan,
                        REPEAS_REQ_INFO *info);
```

Parameters:

`chan` This is the channel number over which the PDU is to be sent.

`info` This pointer to the operation-specific data structure of type **REPEAS_REQ_INFO** contains data specific to the request PDU to be sent.

Return Value:

`MMSREQ_PEND *` This pointer to the request control data structure of type **MMSREQ_PEND** is used to send the PDU. In case of an error, the pointer is set to null and **mms_op_err** is written with the error code.

Corresponding User Confirmation Function: **u_mp_repeas_conf**

Operation-Specific Data Structure Used: **REPEAS_REQ_INFO**

See page 3-123 for a detailed description of this structure.

u_repeas_ind

Usage: This user indication function is called when a ReportEventActionStatus indication is received by a server node. The contents and operation of this function are user-defined, and depend on the application requirements. To conform to the MMS protocol requirements, the application must do one of the following some time after the indication is received:

1) send a positive response using the primitive response function (**`mp_repeas_resp`**) after the specified Event Action status information has been successfully obtained, or

2) call the appropriate primitive negative (error) response function (**`mp_err_resp`**). See **Module 11 — MMS-EASE Error Handling** on page 3-343 for an explanation of this function.

See **Volume 1 — Module 1 — User Indication Function Class** for additional information regarding recommended handling of indications.

Function Prototype: `ST_VOID u_repeas_ind (MMSREQ_IND *ind);`

Parameters:

`ind` This pointer to the **`MMSREQ_IND`** structure is described in **Volume 1 — Module 2 — Request and Indication Control Data Structures**. This pointer must be used to send the response to this indication. The primary item of interest in this structure is **`ind->req_info_ptr`**. This is a pointer to the operation-specific data structure for the ReportEventActionStatus indication (**`REPEAS_REQ_INFO`**). This pointer will always be valid when **`u_repeas_ind`** is called.

Return Value: `ST_VOID` (ignored)

Operation-Specific Data Structure Used: **`REPEAS_REQ_INFO`**

See page 3-123 for a detailed description of this structure.

mp_repeas_resp

Usage: This primitive response function sends a ReportEventActionStatus positive response PDU. It uses the data from a structure of type **REPEAS_RESP_INFO**, pointed to by **info**. This function should be called as a response to the **u_repeas_ind** function being called (a ReportEventActionStatus indication is received), and after the specified Event Action status information has been successfully obtained.

Function Prototype:

```
ST_RET mp_repeas_resp (MMSREQ_IND *ind,
                       REPEAS_RESP_INFO *info);
```

Parameters:

`ind` This pointer to the indication control structure of type **MMSREQ_IND** is received in the **u_repeas_ind** function.

`info` This pointer to an Operation Specific data structure of type **REPEAS_RESP_INFO** contains information specific to the PDU to be sent.

Return Value: `ST_RET` **SD_SUCCESS**. No Error

`<> 0` Error Code

Corresponding User Indication Function: **u_repeas_ind**

Operation-Specific Data Structure Used: **REPEAS_RESP_INFO**

See page 3-123 for a detailed description of this structure.

u_mp_repeas_conf

Usage: This primitive user confirmation function is called when a confirm to a ReportEventActionStatus indication (**mp_repeas**) is received. **resp_err** contains a value indicating whether an error occurred. **resp_info_ptr** may contain data that the remote node sent back in the confirmation or it may contain error information if an error occurred. If no error has occurred (**req->resp_err = CNF_RESP_OK**), ReportEventActionStatus information is available to the application using the **req->resp_info_ptr** pointer. See **Volume 1 — Module 1 — Primitive User Confirmation Function Class** for specific information on the recommended handling of confirms.

Note that the function called for this event may be changed if desired; however, **u_mp_repeas_conf** is the standard default function.

Function Prototype:

```
ST_VOID u_mp_repeas_conf (MMSREQ_PEND *req);
```

Parameters:

`req` This pointer to the **MMSREQ_PEND** structure is returned from the original **mp_repeas** request.

For a positive response (**req->resp_err = CNF_RESP_OK**), the request control structure member **resp_info_ptr** is a pointer to the operation-specific data structure (**REPEAS_RESP_INFO**) for a ReportEventActionStatus.

For a negative response, or other errors, see **Module 11 — MMS-EASE Error Handling** starting on page 3-339 for more information.

Return Value: `ST_VOID` (ignored)

Operation-Specific Data Structure Used: **REPEAS_RESP_INFO**

See page 3-123 for a detailed description of this structure.

23. DefineEventEnrollment Service

This service is used to request that a VMD add the requesting client, or a “third party” client, to the list of users. These users are sent Event Notification service requests. Such Event Notifications result from a transition (or set of transitions) of a corresponding Event Condition object.

Primitive Level DefineEventEnrollment Operations

The following section contains information on how to use the paired primitive interface for the DefineEventEnrollment service. It covers available data structures used by the PPI, and the four primitive level functions that together make up the DefineEventEnrollment service. See **Volume 1 — Module 2 — General Sequence of Events — Confirmed Services** for a detailed explanation of the sequence of events that occur within the MMS-EASE environment during confirmed services.

- The DefineEventEnrollment service consists of the paired primitive functions of **`mp_defee`**, **`u_defee_ind`**, **`mp_defee_resp`**, and **`u_mp_defee_conf`**.

Data Structures

Request/Indication

The operation-specific structure described below is used by the client in issuing a DefineEventEnrollment request (**`mp_defee`**). It is received by the server when a DefineEventEnrollment indication (**`u_defee_ind`**) is received.

```
struct defee_req_info
  {
  OBJECT_NAME evenroll_name;
  OBJECT_NAME evcon_name;
  ST_UCHAR    ec_transitions;
  ST_INT16    alarm_ack_rule;
  ST_BOOLEAN  evact_name_pres;
  OBJECT_NAME evact_name;
  ST_BOOLEAN  client_app_pres;
  ST_INT      client_app_len;
  ST_UCHAR    *client_app;
  ST_BOOLEAN  ackec_name_pres;
  OBJECT_NAME ackec_name;
  SD_END_STRUCT
  };
typedef struct defee_req_info DEFEE_REQ_INFO;
```

Fields:

`evenroll_name` This structure of type **`OBJECT_NAME`** contains the name of the Event Enrollment to be defined.

`evcon_name` This structure of type **`OBJECT_NAME`** contains the name of the Event Condition object corresponding to this Event Enrollment.

`ec_transitions` This bitstring specifies the Event Condition transitions that trigger actions at the server including releasing Event Notifications:

- **0** IDLE-TO-DISABLED
- **1** ACTIVE-TO-DISABLED
- **2** DISABLED-TO-IDLE
- **3** ACTIVE-TO-IDLE
- **4** DISABLED-TO-ACTIVE
- **5** IDLE-TO-ACTIVE
- **6** ANY-TO-DELETED

`alarm_ack_rule` This only exists for Event Enrollments that reference a monitored Event Condition. It indicates the level of acknowledgment required for Event Notification instances generated from this Event Enrollment object. This value also determines whether this Event Enrollment should be included in Alarm Enrollment Summaries. It contains one of the following values:

- **0** **NONE**. This Event Enrollment is NOT included in Alarm Enrollment Summaries. Acknowledgments to Event Notifications in response to these Event Enrollments are allowed, and have no effect on the state of this object.
- **1** **SIMPLE**. Acknowledgment is allowed but not required for Event Notifications specifying a transition to ACTIVE. This has effect on the state of the Event Enrollment object. Acknowledgments to Event Notification specifying a transition to the IDLE state is allowed, and has no effect on the state of this object.
- **2** **ACK-ACTIVE**. Acknowledgment is REQUIRED for notifications specifying transition to the ACTIVE state. Acknowledgment is allowed for transitions to the IDLE state, and has no effect on the state of this object. This Event Enrollment is included in Alarm Enrollment summaries unless it is eliminated by filter criteria.
- **3** **ACK-ALL**. Acknowledgment is REQUIRED FOR ALL notifications specifying transition to the ACTIVE or IDLE states. This is included in the Alarm Enrollment Summaries unless it is eliminated by filter criteria.

NOTE: Acknowledgment is never allowed for event notification specifying a disabled state.

`evact_name_pres` **SD_FALSE**. Do NOT include **evact_name** in the PDU.

SD_TRUE. Include **evact_name** in the PDU.

`evact_name` This structure of type **OBJECT_NAME** contains the name of the corresponding Event Action. Specifying an Event Action name is optional.

`client_app_pres` **SD_FALSE**. Do NOT include **client_app** in the PDU.

SD_TRUE. Include **client_app** in the PDU.

`client_app_len` The length, in bytes, of the data pointed to by **client_app**.

`client_app` This is a pointer to the Client Application Reference. It contains the identification of the enrolled client application.

`ackec_name_pres` **SD_FALSE**. Do NOT include **ackec_name** in the PDU.

SD_TRUE. Include **ackec_name** in the PDU.

`ackec_name` This structure of type **OBJECT_NAME** contains the name of the Acknowledge EventCondition. Specifying an AcknowlegeEventCondition name is optional.

NOTE: For an explanation of the **OBJECT_NAME** structure, see **Volume 1 — Module 2 — MMS Object Name Structure**.

Paired Primitive Interface Functions

mp_defee

Usage: This primitive request function sends a DefineEventEnrollment request PDU to a remote node. It uses the data found in a structure of type **DEFEE_REQ_INFO**, pointed to by **info**. This service is used to add the requesting client (or "third party" client) to the list of users to which EventNotification service requests are sent.

Function Prototype:

```
MMSREQ_PEND *mp_defee (ST_INT chan,
                       DEFEE_REQ_INFO *info);
```

Parameters:

`chan` This is the channel number over which the PDU is to be sent.

`info` This pointer to the operation-specific data structure of type **DEFEE_REQ_INFO** contains data specific to the request PDU to be sent.

Return Value:

`MMSREQ_PEND *` This pointer to the request control data structure of type **MMSREQ_PEND** is used to send the PDU. In case of an error, the pointer is set to null and **mms_op_err** is written with the error code.

Corresponding User Confirmation Function: **u_mp_defee_conf**

Operation-Specific Data Structure Used: **DEFEE_REQ_INFO**

See page 3-129 for a detailed description of this structure.

u_defee_ind

Usage: This user indication function is called when a DefineEventEnrollment indication is received by a server node. The contents and operation of this function are user-defined, and depend on the application requirements. To conform to the MMS protocol requirements, the application must do one of the following some time after the indication is received:

1) send a positive response using the primitive response function (**mp_defee_resp**) after the requesting client has been added to the list of users associated with an EventNotification, or

2) call the appropriate primitive negative (error) response function (**mp_err_resp**). See **Module 11 — MMS-EASE Error Handling** on page 3-343 for an explanation of this function.

See **Volume 1 — Module 1 — User Indication Function Class** for additional information regarding recommended handling of indications.

Function Prototype:

```
ST_VOID u_defee_ind  (MMSREQ_IND *ind);
```

Parameters:

`ind` This pointer to the **MMSREQ_IND** structure is described in **Volume 1 — Module 2 — Request and Indication Control Data Structures**. This pointer must be used to send the response to this indication. The primary item of interest in this structure is **ind->req_info_ptr**. This is a pointer to the operation-specific data structure for a DefineEventEnrollment indication (**DEFEE_REQ_INFO**). This pointer will always be valid when **u_defee_ind** is called.

Return Value: ST_VOID (ignored)

Operation-Specific Data Structure Used: **DEFEE_REQ_INFO**

See page 3-129 for a detailed description of this structure.

mp_defee_resp

Usage: This primitive response function sends a DefineEventEnrollment positive response PDU. This function should be called as a response to the **u_defee_ind** function being called (a DefineEventEnrollment indication is received), and after the requesting client has been added to the list of users associated with an EventNotification.

Function Prototype: `ST_RET mp_defee_resp (MMSREQ_IND *ind);`

Parameters:

`ind` This pointer to the indication control structure of type **MMSREQ_IND** is received in the **u_defee_ind** function.

Return Value: `ST_RET` **SD_SUCCESS**. No Error.

`<> 0` Error Code.

Corresponding User Indication Function: **u_defee_ind**

Operation-Specific Data Structure Used: NONE

u_mp_defee_conf

Usage: This primitive user confirmation function is called when a confirm to a DefineEventEnrollment request (**mp_defee**) is received. **resp_err** contains a value indicating whether an error occurred. **resp_info_ptr** may contain data that the remote node sent back in the confirmation or it may contain error information if an error occurred. See **Volume 1 — Module 1 — Primitive User Confirmation Function Class** for specific information on the recommended handling of confirms.

Note that the function called for this event may be changed if desired; however, **u_mp_defee_conf** is the standard default function.

Function Prototype:

```
ST_VOID u_mp_defee_conf (MMSREQ_PEND *req);
```

Parameters:

`req` This pointer to the **MMSREQ_PEND** structure is returned from the original **mp_defee** request. See **Volume 1 — Module 2 — Request and Indication Control Data Structure** for information on the structure.

For a positive response (**req->resp_err = CNF_RESP_OK**), the request control structure member **resp_info_pres** indicates that there is no confirm data for this function.

For a negative response, or other errors, see **Module 11 — MMS-EASE Error Handling** starting on page 3-339 for more information.

Return Value: `ST_VOID` (ignored)

Operation-Specific Data Structure Used: NONE

24. DeleteEventEnrollment Service

This service is used to request that a VMD delete one or more Event Enrollment objects.

Primitive Level DeleteEventEnrollment Operations

The following section contains information on how to use the paired primitive interface for the DeleteEventEnrollment service. It covers available data structures used by the PPI, and the four primitive level functions that together make up the DeleteEventEnrollment service. See **Volume 1 — Module 2 — General Sequence of Events — Confirmed Services** for a detailed explanation of the sequence of events that occur within the MMS-EASE environment during confirmed services.

- The DeleteEventEnrollment service consists of the paired primitive functions of **`mp_delee`**, **`u_delee_ind`**, **`mp_delee_resp`**, and **`u_mp_delee_conf`**.

Data Structures

Request/Indication

The operation-specific structure described below is used by the client in issuing a DeleteEventEnrollment request (**`mp_delee`**). It is received by the server when a DeleteEventEnrollment indication (**`u_delee_ind`**) is received.

```
struct delee_req_info
  {
  ST_INT16  req_tag;
  union
    {
    OBJECT_NAME evcon_name;
    OBJECT_NAME evact_name;
    ST_INT      num_of_names;
    } sod;
/*OBJECT_NAME   name_list [num_of_names];                  */
  SD_END_STRUCT
  };
typedef struct delee_req_info DELEE_REQ_INFO;
```

Fields:

`req_tag` This indicates the scope of delete request by specifying the extent of the delete:

- **`0`** **SPECIFIC**. Request deletion of the specific Event Enrollment object listed in the Event Enrollment names.
- **`1`** **EVENT CONDITION**. Request deletion of all the Event Enrollment referenced by the Event Condition object specified in the Event Condition name.
- **`2`** **EVENT ACTION**. Request deletion of all the Event Enrollment referenced by the Event Action object specified in the Event Action name.

`evcon_name` This structure of type **`OBJECT_NAME`** contains the name of the Event Condition that references the Event Enrollments to be deleted. Use only if **`req_tag = 1`**.

`evact_name` This structure of type **`OBJECT_NAME`** contains the name of the Event Action which references the Event Enrollments to be deleted. Use only if **`req_tag = 2`**.

`num_of_names` This indicates the number of names of Event Enrollment objects that are to be deleted. Use only if **`req_tag = 0`**.

`name_list` This list of structures of type **`OBJECT_NAME`** contains the names of the Event Enrollment objects to be deleted.

NOTES:

1. For an explanation of the **`OBJECT_NAME`** structure, see **Volume 1 — Module 2 — MMS Object Name Structure**.
2. FOR REQUEST ONLY, when allocating memory for the **`DELEE_REQ_INFO`** structure, enough memory must be allocated to hold the actual list of Event Enrollment names contained in **`name_list`**. The following C statement can be used:

```
info = (DELEE_REQ_INFO *) chk_malloc (sizeof (DELEE_REQ_INFO) + (num_of_names *
            sizeof(OBJECT_NAME)));
```

Response/Confirm

The operation-specific data structure described below is used by the server in issuing a DeleteEventEnrollment response (**`mp_delee_resp`**). It is received by the client when a DeleteEventEnrollment confirm (**`u_mp_delee_conf`**) is received.

```
struct delee_resp_info
  {
  ST_UINT32   cand_not_deleted;
  };
typedef struct delee_resp_info DELEE_RESP_INFO;
```

Parameters:

`cand_not_deleted` This specifies the number of candidates that were not deleted as a result of the Delete EventEnrollment request.

Paired Primitive Interface Functions

mp_delee

Usage: This primitive request function sends a DeleteEventEnrollment request PDU to a remote node. It uses the data found in a structure of type **DELEE_REQ_INFO**, pointed to by **info**. This service is used to delete one or more Event Enrollment objects.

Function Prototype:

```
MMSREQ_PEND *mp_delee (ST_INT chan,
                       DELEE_REQ_INFO *info);
```

Parameters:

chan	This is the channel number over which the PDU is to be sent.
info	This pointer to the operation-specific data structure of type **DELEE_REQ_INFO** contains data specific to the request PDU to be sent.

Return Value:

MMSREQ_PEND *	This pointer to the request control data structure of type **MMSREQ_PEND** is used to send the PDU. In case of an error, the pointer is set to null and **mms_op_err** is written with the error code.

Corresponding User Confirmation Function: **u_mp_delee_conf**

Operation-Specific Data Structure Used: **DELEE_REQ_INFO**

See page 3-135 for a detailed description of this structure.

u_delee_ind

Usage: This user indication function is called when a DeleteEventEnrollment indication is received by a server node. The contents and operation of this function are user-defined, and depend on the application requirements. To conform to the MMS protocol requirements, the application must do one of the following some time after the indication is received:

1) send a positive response using the primitive response function (**mp_delee_resp**) after the specified Event Enrollment objects have been deleted, or

2) call the appropriate primitive negative (error) response function (**mp_err_resp**) within the user indication function. See **Module 11 — MMS-EASE Error Handling** on page 3-343 for an explanation of this function.

See **Volume 1 — Module 1 — User Indication Function Class** for additional information regarding recommended handling of indications.

Function Prototype:

```
ST_VOID u_delee_ind  (MMSREQ_IND *ind);
```

Parameters:

`ind` This pointer to the **MMSREQ_IND** structure is described in **Volume 1 — Module 2 — Request and Indication Control Data Structures**. This pointer must be used to send the response to this indication. The primary item of interest in this structure is **ind->req_info_ptr**. This is a pointer to the operation-specific data structure for the DeleteEventEnrollment indication (**DELEE_REQ_INFO**). This pointer will always be valid when **u_delee_ind** is called.

Return Value: `ST_VOID` (ignored)

Operation-Specific Data Structure Used: **DELEE_REQ_INFO**

See page 3-135 for a detailed description of this structure.

mp_delee_resp

Usage: This primitive response function sends a DeleteEventEnrollment positive response PDU using the data from a structure of type **DELEE_RESP_INFO**, pointed to by **info**. This function should be called as a response to the **u_delee_ind** function being called (a DeleteEvent Enrollment indication is received), and after the specified Event Enrollment objects have been deleted.

Function Prototype:

```
ST_RET mp_delee_resp (MMSREQ_IND *ind,
                      DELEE_RESP_INFO *info);
```

Parameters:

`ind` This pointer to the indication control structure of type **MMSREQ_IND** is received in the **u_delee_ind** function.

`info` This pointer to an Operation Specific data structure of type **DELEE_RESP_INFO** contains information specific to the PDU to be sent.

Return Value: `ST_RET`

SD_SUCCESS. No Error.

`<> 0` Error Code.

Corresponding User Indication Function: **u_delee_ind**

Operation-Specific Data Structure Used: **DELEE_RESP_INFO**

See page 3-136 for a detailed description of this structure.

u_mp_delee_conf

Usage: This primitive user confirmation function is called when a confirm to a DeleteEventEnrollment request (**mp_delee**) is received. **resp_err** contains a value indicating whether an error occurred. **resp_info_ptr** may contain data that the remote node sent back in the confirmation or it may contain error information if an error occurred. If no error has occurred (**req->resp_err = CNF_RESP_OK**), DeleteEventEnrollment information is available to the application using the **req->resp_info_ptr** pointer. See **Volume 1 — Module 1 — Primitive User Confirmation Function Class** for specific information on the recommended handling of confirms.

Note that the function called for this event may be changed if desired; however, **u_mp_delee_conf** is the standard default function.

Function Prototype:

```
ST_VOID u_mp_delee_conf (MMSREQ_PEND *req);
```

Parameters:

`req` This pointer to the **MMSREQ_PEND** structure is returned from the original **mp_delee** request. See **Volume 1 — Module 2 — Request and Indication Control Data Structures** for information on the structure.

For a positive response (**req->resp_err = CNF_RESP_OK**), the request control structure member **resp_info_ptr** is a pointer to the operation-specific data structure (**DELEE_RESP_INFO**) for the DeleteEventEnrollment function.

For a negative response, or other errors, see **Module 11 — MMS-EASE Error Handling** starting on page 3-339 for more information.

Return Value: `ST_VOID` (ignored)

Operation-Specific Data Structure Used: **DELEE_RESP_INFO**

See page 3-136 for a detailed description of this structure.

25. GetEventEnrollmentAttributes Service

This service is used to request that a VMD return the descriptive attributes. These can be of an Event Enrollment object or a list of Event Enrollment objects satisfying a set of criteria.

Primitive Level GetEventEnrollmentAttributes Operations

The following section contains information on how to use the paired primitive interface for the GetEventEnrollmentAttributes service. It covers available data structures used by the PPI, and the four primitive level functions that together make up the GetEventEnrollmentAttributes service. See **Volume 1 — Module 2 — General Sequence of Events — Confirmed Services** for a detailed explanation of the sequence of events that occur within the MMS-EASE environment during confirmed services.

- The GetEventEnrollmentAttributes service consists of the paired primitive functions of **`mp_geteea`**, **`u_geteea_ind`**, **`mp_geteea_resp`**, and **`u_mp_geteea_conf.`**

Data Structures

Request/Indication

The operation-specific structure described below is used by the client in issuing a GetEventEnrollmentAttributes request (**`mp_geteea`**). It is received by the server when a GetEventEnrollmentAttributes indication (**`u_geteea_ind`**) is received.

```
struct geteea_req_info
  {
  ST_INT16    scope_of_req;
  ST_BOOLEAN  client_app_pres;
  ST_INT      client_app_len;
  ST_UCHAR    *client_app;
  ST_BOOLEAN  evcon_name_pres;
  OBJECT_NAME evcon_name;
  ST_BOOLEAN  evact_name_pres;
  OBJECT_NAME evact_name;
  ST_BOOLEAN  ca_name_pres;
  OBJECT_NAME ca_name;
  ST_BOOLEAN  eenames_pres;
  ST_INT      num_of_eenames;
/*OBJECT_NAME name_list [num_of_eenames];           */
  SD_END_STRUCT
  };
typedef struct geteea_req_info GETEEA_REQ_INFO;
```

Fields:

`scope_of_req` This indicates the scope of the request for Event Enrollment objects. It specifies one of four values:

- **`0`** **SPECIFIC**. Specifies the attributes for a list of Event Enrollment objects, identified in Event Enrollment names (name_list[num_of_eenames]).
- **`1`** **CLIENT**. Specifies that a list of all Event Enrollment objects for which the specified client (**`client_app`**) is enrolled.

2 **EVENT CONDITION**. Specifies that a list of all Event Enrollment objects for the specified Event Condition name (**evcon_name**).

3 **EVENT ACTION**. Specifies that a list of all Event Enrollment objects for the specified Event Action name (**evact_name**).

`client_app_pres` **SD_FALSE**. Do NOT include **client_app** in the PDU.

SD_TRUE. Include **client_app** in the PDU.

`client_app_len` This is the length, in bytes, of the data pointed to by **client_app**.

`client_app` This is a pointer to the Client Application Reference containing the identification of the enrolled client application. Use only if **scope_of_req = 1**.

`evcon_name_pres` **SD_FALSE**. Do NOT include **evcon_name** in the PDU.

SD_TRUE. Include **evcon_name** in the PDU.

`evcon_name` This structure of type **OBJECT_NAME** contains the name of the event condition for which the attributes are being requested. Use only if **scope_of_req = 2**.

`evact_name_pres` **SD_FALSE**. Do NOT include **evact_name** in the PDU.

SD_TRUE. Include **evact_name** in the PDU.

`evact_name` This structure of type **OBJECT_NAME** contains the name of the event action for which the attributes are being requested. Use only if **scope_of_req = 3**.

`ca_name_pres` **SD_FALSE**. Do NOT include **ca_name** in the PDU. Begin the name list response from the beginning of the list.

SD_TRUE. Include **ca_name** in the PDU. Use this when you must make multiple requests to obtain the entire name list because the entire list of names will not fit in a single response.

`ca_name` This is a pointer to the name of the object after which the list should begin. If this field is not included, the list will begin with the first name in the list. **ca_name** should be specified when the request is a subsequent request to the original request because the entire name list could not fit into a single response.

`eenames_pres` **SD_FALSE**. Do NOT include **num_of_eenames** and **name_list** in the PDU.

SD_TRUE. Include **num_of_eenames** and **name_list** in the PDU.

`num_of_eenames` This indicates the number of Event Enrollment names present in the list of Event Enrollments. Use only if **scope_of_req = 0**.

`name_list` This array of structures of type **OBJECT_NAME** contains the list of Event Enrollment names whose attributes are to be obtained. Use only if **scope_of_req = 0**.

NOTES:

1. For information on the **OBJECT_NAME** structure, see **Volume 1 — Module 2 — MMS Object Name Structure**.
2. When allocating memory for the **GETEEA_REQ_INFO** structure, enough memory must be allocated to hold the actual list of **OBJECT_NAMES** contained in **name_list**. The following C statement can be used:

```
info = (GETEEA_REQ_INFO *) chk_malloc(sizeof (GETEEA_REQ_INFO) + (num_of_eenames
            * sizeof(OBJECT_NAME)));
```

Response/Confirm

The operation-specific data structure described below is used by the server in issuing a GetEventEnrollmentAttributes response (**mp_geteea_resp**). It is received by the client when a GetEventEnrollmentAttributes confirm (**u_mp_geteea_conf**) is received.

```
struct geteea_resp_info
  {
  ST_BOOLEAN       more_follows;
  ST_INT           num_of_evenroll;
/*EVENT_ENROLLMENT evenroll_list [num_of_evenroll]; */
  SD_END_STRUCT
  };
typedef struct geteea_resp_info  GETEEA_RESP_INFO;
```

Fields:

`more_follows` **SD_TRUE**. There are more event enrollments available.

SD_FALSE. This is the end of the event enrollment list.

`num_of_evenroll` This indicates the number of event enrollments present in this event enrollment list.

`evenroll_list` This array of structures of type **EVENT_ENROLLMENT** contains a list of event enrollments from which to obtain attributes. See below for information on this structure.

NOTE: When allocating memory for the **GETEEA_RESP_INFO** structure, enough memory must be allocated to hold the actual list of event enrollment structures contained in **evenroll_list**. The following C statement can be used:

```
info = (GETEEA_RESP_INFO *) chk_malloc(sizeof (GETEEA_RESP_INFO) +
             (num_of_evenroll * sizeof(EVENT_ENROLLMENT)));
```

```
struct event_enrollment
  {
  OBJECT_NAME evenroll_name;
  ST_CHAR     evcon_name_tag;
  OBJECT_NAME evcon_name;
  ST_BOOLEAN  evact_name_pres;
  ST_CHAR     evact_name_tag;
  OBJECT_NAME evact_name;
  ST_BOOLEAN  client_app_pres;
  ST_INT      client_app_len;
  ST_UCHAR    *client_app;
  ST_BOOLEAN  mms_deletable;
  ST_INT16    ee_class;
  ST_INT16    duration;
  ST_BOOLEAN  invoke_id_pres;
  ST_UINT32   invoke_id;
  ST_BOOLEAN  rem_acc_delay_pres;
  ST_UINT32   rem_acc_delay;
  ST_BOOLEAN  addl_detail_pres;
  ST_INT      addl_detail_len;
  ST_UCHAR    *addl_detail;
  ST_BOOLEAN  ackec_name_pres;
  ST_CHAR     ackec_name_tag;
  OBJECT_NAME ackec_name;
  SD_END_STRUCT
  };
typedef struct event_enrollment  EVENT_ENROLLMENT;
```

Fields:

`evenroll_name` This structure of type **OBJECT_NAME** contains the name of the event enrollment for which this is intended.

`evcon_name_tag` This is a tag indicating whether there is an event condition name for this event enrollment:

SD_FALSE. The event condition still exists. The event condition name, specified by **evcon_name**, will be included in the PDU.

SD_TRUE. The underlying event condition has been deleted. As a result, the event condition name (**evcon_name**) will NOT be included in the PDU. The event condition name is UNDEFINED.

`evcon_name` This is a structure of type **OBJECT_NAME** containing the name of the Event Condition for which the status is to be obtained.

`evact_name_pres` **SD_FALSE**. Do NOT include **evact_name** in the PDU.

SD_TRUE. Include **evact_name** in the PDU.

`evact_name_tag` A tag indicating whether there is an Event Action name for this event enrollment:

SD_TRUE. The Event Action name is defined. **evact_name** will be included in the PDU.

SD_FALSE. The Event Action name is UNDEFINED. Do not include **evact_name** in the PDU.

`evact_name` This structure of type **OBJECT_NAME** contains the name of the Event Action. Used only if the Event Enrollment is contained in an Event Action name.

`client_app_pres` **SD_FALSE**. Do NOT include **client_app** in the PDU.

SD_TRUE. Include **client_app** in the PDU.

`client_app_len` This is the length, in bytes, of the data pointed to by **client_app**.

`client_app` This pointer to the Client Application Reference contains the identification of the enrolled client application.

`mms_deletable` **SD_FALSE**. This Event Enrollment is NOT deletable using a service request.

SD_TRUE. This Event Enrollment is deletable using a service request.

`ee_class` This is the event enrollment class:

- **0** **MODIFIER**. The Event Enrollment is temporary (executed one time and deleted). It is created as a result of a received service indication that specifies a confirmed MMS service modified by the **ATTACH_TO_EVCON** modifier. See page 3-184 for an explanation of modifier handling.
- **1** **NOTIFICATION**. The Event Enrollment is explicitly defined or predefined. The EventNotification requests (**mp_evnot**) should be issued when any of the indicated Event Condition occurrences transition.

`duration` This indicates the duration of the Event Enrollment:

- **0** **CURRENT**. This indicates that this Event Enrollment is specified for the life of the application association over which this Event Enrollment object was defined. This is the default.
- **1** **PERMANENT**. This indicates that this Event Enrollment is specified for the life of the VMD unless explicitly defined.

`invoke_id_pres` **SD_FALSE**. Do NOT include **invoke_id** in the PDU.

SD_TRUE. Include **invoke_id** in the PDU.

`invoke_id` This contains the invoke ID of the transaction object's modified service. Use only if **ee_class = 0** (modifier).

`rem_acc_delay_pres` **SD_FALSE**. Do NOT include **rem_acc_delay** in the PDU.

SD_TRUE. Include **rem_acc_delay** in the PDU.

`rem_acc_delay` This indicates the remote acceptable delay. This is used only if **ee_class = 0** (modifier). Its value can either be FOREVER, or the remaining duration of time in seconds for which the requester is willing to wait for the specified Event Condition to occur.

If more than one **ATTACH_TO_EVCON** modifier is specified for an Event Condition, this indicates individually the remaining acceptable delay for each modifier following the beginning of processing. If this equals zero, the procedure for Event Enrollment deletion should be followed.

`addl_detail_pres` **SD_FALSE**. Do NOT include **addl_detail** in the PDU.

SD_TRUE. Include **addl_detail** in the PDU.

`addl_detail_len` The length, in bytes, of the data pointed to by **addl_detail**.

`addl_detail` This contains Additional Detail. The content and form of this data must be valid ASN.1.

`ackec_name_pres` **SD_FALSE**. Do NOT include **ackec_name** in the PDU.

SD_TRUE. Include **ackec_name** in the PDU.

`ackec_name_tag` This is a tag indicating whether there is an acknowledge event condition name for this event enrollment:

SD_FALSE. The acknowledge event condition still exists. The Acknowledge EventCondition name, specified by **ackec_name**, will be included in the PDU.

SD_TRUE. The underlying acknowledge event condition has been deleted. As a result, the AcknowledgeEventCondition name (**ackec_name**) will NOT be included in the PDU.

`ackec_name` This structure of type **OBJECT_NAME** contains the name of the AcknowledgeEventCondition. Used only if the Event Enrollment is contained in an AcknowledgeEventCondition name.

Paired Primitive Interface Functions

mp_geteea

Usage: This primitive request function sends a GetEventEnrollmentAttributes request PDU to a remote node. It uses the data found in a structure of type **GETEEA_REQ_INFO**, pointed to by **info**. This service is used to obtain a list of the attributes of the specified Event Enrollment(s).

Function Prototype:

```
MMSREQ_PEND *mp_geteea (ST_INT chan,
                        GETEEA_REQ_INFO *info);
```

Parameters:

chan	This is the channel number over which the PDU is to be sent.
info	This pointer to the operation-specific data structure of type **GETEEA_REQ_INFO** contains data specific to the request PDU to be sent.

Return Value:

MMSREQ_PEND *	This pointer to the request control data structure of type **MMSREQ_PEND** is used to send the PDU. In case of an error, the pointer is set to null and **mms_op_err** is written with the error code.

Corresponding User Confirmation Function: **u_mp_geteea_conf**

Operation-Specific Data Structure Used: **GETEEA_REQ_INFO**

See page 3-141 for a detailed description of this structure.

u_geteea_ind

Usage: This user indication function is called when a GetEventEnrollmentAttributes indication is received by a server node. The contents and operation of this function are user-defined, and depend on the application requirements. To conform to the MMS protocol requirements, the application must do one of the following some time after the indication is received:

1) send a positive response using the primitive response function (**`mp_geteea_resp`**) after the specified event enrollment attributes have been successfully obtained, or

2) call the appropriate primitive negative (error) response function (**`mp_err_resp`**). See **Module 11 — MMS-EASE Error Handling** on page 3-343 for an explanation of this function.

See **Volume 1 — Module 1 — Primitive User Confirmation Function Class** for additional information regarding the recommended handling of indications.

Function Prototype: `ST_VOID u_geteea_ind (MMSREQ_IND *ind);`

Parameters:

`ind` This pointer to the **`MMSREQ_IND`** structure is described in **Volume 1 — Module 2 — Request and Indication Control Data Structures**. This pointer must be used to send the response to this indication. The primary item of interest in this structure is **`ind->req_info_ptr`**. This is a pointer to the operation-specific data structure for a GetEventEnrollmentAttributes indication (**`GETEEA_REQ_INFO`**). This pointer will always be valid when **`u_geteea_ind`** is called.

Return Value: `ST_VOID` (ignored)

Operation-Specific Data Structure Used: **`GETEEA_REQ_INFO`**

See page 3-141 for a detailed description of this structure.

mp_geteea_resp

Usage: This primitive response function sends a GetEventEnrollmentAttributes positive response PDU. It uses the data from a structure of type **GETEEA_RESP_INFO**, pointed to by **info**. This function should be called as a response to the **u_geteea_ind** function being called (a GetEventEnrollmentAttributes indication is received), and after the specified event enrollment attributes have been successfully obtained.

Function Prototype:

```
ST_RET mp_geteea_resp (MMSREQ_IND *ind,
                       GETEEA_RESP_INFO *info);
```

Parameters:

`ind` This pointer to the indication data structure of type **MMSREQ_IND** is passed to the **u_geteea_ind** function when it was called.

`info` This pointer to an Operation-Specific data structure of type **GETEEA_RESP_INFO** contains information specific to the response PDU to be sent.

Return Value: `ST_RET` **SD_SUCCESS**. No Error.

`<> 0` Error Code.

Corresponding User Indication Function: **u_geteea_ind**

Operation-Specific Data Structure Used: **GETEEA_RESP_INFO**

See page 3-143 for a detailed description of this structure.

u_mp_geteea_conf

Usage: This primitive user confirmation function is called when a confirm to a GetEventEnrollment Attributes request (**mp_geteea**) is received. **resp_err** contains a value indicating whether an error occurred. **resp_info_ptr** may contain data that the remote node sent back in the confirmation or it may contain error information if an error occurred. If no error has occurred (**req->resp_err = CNF_RESP_OK**), GetEventEnrollmentAttributes information is available to the application using the **req->resp_info_ptr** pointer. See **Volume 1 — Module 1 — Primitive User Confirmation Function Class** for specific information on the recommended handling of confirms.

Note that the function called for this event may be changed if desired; however, **u_mp_geteea_conf** is the standard default function.

Function Prototype:

```
ST_VOID u_mp_geteea_conf (MMSREQ_PEND *req);
```

Parameters:

`req` This pointer to the **MMSREQ_PEND** structure is returned from the original **mp_geteea** request. See **Volume 1 — Module 2 — Request and Indication Control Data Structures** for information on the structure.

For a positive response (**req->resp_err = CNF_RESP_OK**), the request control structure member **resp_info_ptr** is a pointer to the operation-specific data structure (**GETEEA_RESP_INFO**) for the GetEventEnrollmentAttributes function.

For a negative response, or other errors, see **Module 11 — MMS-EASE Error Handling** starting on page 3-339 for more information.

Return Value: `ST_VOID` (ignored)

Operation-Specific Data Structure Used: **GETEEA_RESP_INFO**

See page 3-143 for a detailed description of this structure.

26. ReportEventEnrollmentStatus Service

This service is used to obtain the status of a single notification Event Enrollment object from a VMD.

Primitive ReportEventEnrollmentStatus Operations

The following section contains information on how to use the paired primitive interface for the ReportEventEnrollmentStatus service. It covers available data structures used by the PPI, and the four primitive level functions that together make up the ReportEventEnrollmentStatus service. See **Volume 1 — Module 2 — General Sequence of Events — Confirmed Services** for a detailed explanation of the sequence of events that occur within the MMS-EASE environment during confirmed services.

- The ReportEventEnrollmentStatus service consists of the paired primitive functions of **`mp_repees`**, **`u_repees_ind`**, **`mp_repees_resp`**, and **`u_mp_repees_conf`**.

Data Structures

Request/Indication

The operation-specific structure described below is used by the client in issuing a ReportEventEnrollmentStatus request (**`mp_repees`**). It is received by the server when a ReportEventEnrollmentStatus indication (**`u_repees_ind`**) is received.

```
struct repees_req_info
  {
  OBJECT_NAME  evenroll_name;
  };
typedef struct repees_req_info REPEES_REQ_INFO;
```

Fields:

`evenroll_name` This structure of type **`OBJECT_NAME`** contains the name of the Event Enrollment for which the status is to be obtained. See **Volume 1 — Module 2 — MMS Object Name Structure** for an explanation of this structure.

Response/Confirm

The operation-specific data structure described below is used by the server in issuing a ReportEventEnrollmentStatus response (**`mp_repees_resp`**). It is received by the client when a ReportEventEnrollmentStatus confirm (**`u_mp_repees_conf`**) is received.

```
struct repees_resp_info
  {
  ST_UCHAR   ec_transitions;
  ST_UCHAR   not_lost;
  ST_INT16   duration;
  ST_BOOLEAN alarm_ack_rule_pres;
  ST_INT16   alarm_ack_rule;
  ST_INT16   cur_state;
  };
typedef struct repees_resp_info  REPEES_RESP_INFO;
```

Fields:

`ec_transitions` This bitstring indicates the transitions that trigger actions at the server:

- **0** IDLE-TO-DISABLED
- **1** ACTIVE-TO-DISABLED
- **2** DISABLED-TO-IDLE
- **3** ACTIVE-TO-IDLE
- **4** DISABLED-TO-ACTIVE
- **5** IDLE-TO-ACTIVE
- **6** ANY-TO-DELETED

`not_lost` **SD_FALSE**. There is no difficulty. Event Notifications are being sent. This is the default.

SD_TRUE. The last Event Notifications was not able to be generated.

`duration` This indicates the duration of the Event Enrollment:

- **0** **CURRENT**. This indicates that this Event Enrollment is specified for the life of the application association over which this Event Enrollment object was defined. This is the default.
- **1** **PERMANENT**. This indicates that this Event Enrollment is specified for the life of the VMD unless explicitly defined.

`alarm_ack_rule_pres` **SD_FALSE**. Do NOT include **alarm_ack_rule** in the PDU.

SD_TRUE. Include **alarm_ack_rule** in the PDU.

`alarm_ack_rule` This only exists for Event Enrollments that reference a monitored Event Condition. It indicates the level of acknowledgment required for Event Notification instances generated as a result of this Event Enrollment object. This value also determines, whether this Event Enrollment should be included in Alarm Enrollment Summaries. It contains one of the following values:

- **0** **NONE**. This Event Enrollment is NOT included in Alarm Enrollment Summaries. Acknowledgments to Event Notifications in response to these Event Enrollments are allowed, and have no effect on the state of this object.
- **1** **SIMPLE**. Acknowledgment is allowed but not required for Event Notifications specifying a transition to ACTIVE. This has effect on the state of the Event Enrollment object. Acknowledgments to Event Notification specifying a transition to the IDLE state is allowed, and has no effect on the state of this object.
- **2** **ACK-ACTIVE**. Acknowledgment is REQUIRED for notifications specifying transition to the ACTIVE state. Acknowledgment is allowed for transitions to the IDLE state, and has no effect on the state of this object. This Event Enrollment is included in Alarm Enrollment summaries unless filter criteria eliminates it.
- **3** **ACK-ALL**. Acknowledgment is REQUIRED FOR ALL notifications specifying transition to the ACTIVE or IDLE states. This is included in the Alarm Enrollment Summaries unless it is eliminated by filter criteria.

NOTE: Acknowledgment is never allowed for event notification specifying a disabled state.

`cur_state` This indicates the current state of the event condition:

`0` **DISABLED**.

`1` **IDLE**.

`2` **ACTIVE**.

`3` **NO-ACK-A**. Indicates that required acknowledgment for a transition of the Event Condition to the ACTIVE state has not yet been received from the client.

If the major state is **ACTIVE**, then this is the minor state. Its value of the Time Active Acknowledge (**`aack_time`**) is either UNDEFINED or is earlier than the value of the Time of Last Transition to Active (**`tta_time`**).

If the major state is **IDLE**, then this is the minor state. Its value of **`tta_time`** is not equal to UNDEFINED, and that the value of **`aack_time`** is either UNDEFINED, or earlier in time than the value of **`tta_time.`**

`4` **NO-ACKED-I**. Indicates that a required acknowledgment for a transition of the Event Condition to the IDLE state has not yet been received from the client.

This minor state is only defined for a major **IDLE** state specifying a value of Alarm Acknowledgment Rule (**`alarm_ack_rule`**) equal to **ACK-ALL**. It indicates that the required acknowledgment for a previous transition to **ACTIVE** state has been received. The value of Time IDLE Acknowledged (**`iack_time`**) is UNDEFINED or earlier than the value of Time of Last Transition to IDLE (**`tti_time`**).

`5` **ACKED**. Indicates that all required acknowledgments have been received from the client.

Paired Primitive Interface Functions

mp_repees

Usage: This primitive request function sends a ReportEventEnrollmentStatus request PDU to a remote node. It uses the data found in a structure of type **REPEES_REQ_INFO**, pointed to by **info**. This service is used to obtain the status of a single notification Event Enrollment object from a remote node.

Function Prototype:

```
MMSREQ_PEND *mp_repees (ST_INT chan,
                        REPEES_REQ_INFO *info);
```

Parameters:

chan	This is the channel number over which the PDU is to be sent.
info	This pointer to the operation-specific data structure of type **REPEES_REQ_INFO** contains data specific to the request PDU to be sent.

Return Value:

MMSREQ_PEND *	This pointer to the request control data structure of type **MMSREQ_PEND** is used to send the PDU. In case of an error, the pointer is set to null and **mms_op_err** is written with the error code.

Corresponding User Confirmation Function: **u_mp_repees_conf**

Operation-Specific Data Structure Used: **REPEES_REQ_INFO**

See page 3-151 for a detailed description of this structure.

u_repees_ind

Usage: This user indication function is called when a ReportEventEnrollmentStatus indication is received by a server node. The contents and operation of this function are user-defined, and depend on the application requirements. To conform to the MMS protocol requirements, the application must do one of the following some time after the indication is received:

1) send a positive response using the primitive response function (**`mp_repees_resp`**) after the specified Event Enrollment information has been successfully obtained, or

2) call the appropriate primitive negative (error) response function (**`mp_err_resp`**). See **Module 11 — MMS-EASE Error Handling** on page 3-343 for an explanation of this function.

See **Volume 1 — Module 1 — Primitive User Confirmation Function Class** for additional information regarding recommended handling of indications.

Function Prototype: `ST_VOID u_repees_ind (MMSREQ_IND *ind);`

Parameters:

`ind` This pointer to the **`MMSREQ_IND`** structure is described in **Volume 1 — Module 2 — Request and Indication Control Data Structures**. This pointer must be used to send the response to this indication. The primary item of interest in this structure is **`ind->req_info_ptr`**. This is a pointer to the operation-specific data structure for a ReportEventEnrollmentStatus indication (**`REPEES_REQ_INFO`**). This pointer will always be valid when **`u_repees_ind`** is called.

Return Value: `ST_VOID` (ignored)

Operation-Specific Data Structure Used: **`REPEES_REQ_INFO`**

See page 3-151 for a detailed description of this structure.

mp_repees_resp

Usage: This primitive response function sends a ReportEventEnrollmentStatus positive response PDU using the data from a structure of type **REPEES_RESP_INFO**, pointed to by **info**. This function should be called as a response to the **u_repees_ind** function being called (a ReportEventEnrollmentStatus indication is received), and after the specified Event Enrollment information has been successfully obtained.

Function Prototype:

```
ST_RET mp_repees_resp (MMSREQ_IND *ind,
                       REPEES_RESP_INFO *info);
```

Parameters:

`ind` This pointer to the indication control structure of type **MMSREQ_IND** is received in the **u_repees_ind** function.

`info` This pointer to an Operation-Specific data structure of type **REPEES_RESP_INFO** contains information specific to the response PDU being sent.

Return Value: `ST_RET` **SD_SUCCESS**. No Error.

`<> 0` Error Code.

Corresponding User Indication Function: **u_repees_ind**

Operation-Specific Data Structure Used: **REPEES_RESP_INFO**

See page 3-151 for a detailed description of this structure.

u_mp_repees_conf

Usage: This primitive user confirmation function is called when a confirm to a ReportEventEnrollmentStatus request (**mp_repees**) is received. **resp_err** contains a value indicating whether an error occurred. **resp_info_ptr** may contain data that the remote node sent back in the confirmation or it may contain error information if an error occurred. If no error has occurred (**req->resp_err = CNF_RESP_OK**), ReportEventEnrollmentStatus information is available to the application using the **req->resp_info_ptr** pointer. See **Volume 1 — Module 1 — Primitive User Confirmation Function Class** for specific information on the recommended handling of confirms.

Note that the function called for this event may be changed if desired; however, **u_mp_repees_conf** is the standard default function.

Function Prototype:

```
ST_VOID u_mp_repees_conf (MMSREQ_PEND *req);
```

Parameters:

`req` This pointer to the **MMSREQ_PEND** structure is returned from the original **mp_repees** request. See **Volume 1 — Module 2 — Request and Indication Control Data Structures** for information on the structure.

For a positive response (**req->resp_err = CNF_RESP_OK**), the request control structure member **resp_info_ptr** is a pointer to the operation-specific data structure (**REPEES_RESP_INFO**) for a ReportEventEnrollmentStatus function.

For a negative response, or other errors, see **Module 11 — MMS-EASE Error Handling** starting on page 3-323 for more information.

Return Value: `ST_VOID` (ignored)

Operation-Specific Data Structure Used: **REPEES_RESP_INFO**

See page 3-151 for a detailed description of this structure.

27. AlterEventEnrollment Service

This service is used to request that a VMD replace one or both of the attributes of an existing Event Enrollment object. These attributes are Event Condition Transitions (**ec_transitions**) and Alarm Acknowledgment Rule (**alarm_ack_rule**). This object has the value of the Event Condition class (**ec_class**) equal to MONITORED. This service cannot be used to alter a modifier Event Enrollment object.

Primitive Level AlterEventEnrollment Operations

The following section contains information on how to use the paired primitive interface for the AlterEventEnrollment service. It covers available data structures used by the PPI, and the four primitive level functions that together make up the AlterEventEnrollment service. See **Volume 1 — Module 2 — General Sequence of Events — Confirmed Services** for a detailed explanation of the sequence of events that occur within the MMS-EASE environment during confirmed services.

- The AlterEventEnrollment service consists of the paired primitive functions of **mp_altee**, **u_altee_ind**, **mp_altee_resp**, and **u_mp_altee_conf**.

Data Structures

Request/Indication

The operation-specific structure described below is used by the client in issuing an AlterEventEnrollment request (**mp_altee**). It is received by the server when an AlterEventEnrollment indication (**u_altee_ind**) is received.

```
struct altee_req_info
  {
  OBJECT_NAME  evenroll_name;
  ST_BOOLEAN ec_transitions_pres;
  ST_UCHAR   ec_transitions;
  ST_BOOLEAN alarm_ack_rule_pres;
  ST_INT16   alarm_ack_rule;
  SD_END_STRUCT
  };
typedef struct altee_req_info ALTEE_REQ_INFO;
```

Fields:

`evenroll_name` This structure of type **OBJECT_NAME** contains the name of the Event Enrollment for which the attributes are to be altered. See **Volume 1 — Module 2 — MMS Object Name Structure** for an explanation of this structure.

`ec_transitions_pres`

SD_FALSE. Do NOT include **ec_transitions** in the PDU.

SD_TRUE. Include **ec_transitions** in the PDU.

`ec_transitions` This bitstring indicates the transitions that trigger actions at the server:

0	IDLE-TO-DISABLED
1	ACTIVE-TO-DISABLED
2	DISABLED-TO-IDLE
3	ACTIVE-TO-IDLE
4	DISABLED-TO-ACTIVE
5	IDLE-TO-ACTIVE
6	ANY-TO-DELETED

`alarm_ack_rule_pres`

`SD_FALSE`. Do NOT include **`alarm_ack_rule`** in the PDU.

`SD_TRUE`. Include **`alarm_ack_rule`** in the PDU.

`alarm_ack_rule` This only exists for Event Enrollments that reference a monitored Event Condition. It indicates the level of acknowledgment required for Event Notification instances generated as a result of this Event Enrollment object. This value also determines, whether this Event Enrollment should be included in Alarm Enrollment Summaries. It contains one of the following values:

`0` **NONE**. This Event Enrollment is NOT included in Alarm Enrollment Summaries. Acknowledgments to Event Notifications in response to these Event Enrollments are allowed, and have no effect on the state of this object.

`1` **SIMPLE**. Acknowledgment is allowed but not required for Event Notifications specifying a transition to ACTIVE. This has effect on the state of the Event Enrollment object. Acknowledgments to Event Notification specifying a transition to the IDLE state is allowed, and has no effect on the state of this object.

`2` **ACK-ACTIVE**. Acknowledgment is REQUIRED for notifications specifying transition to the ACTIVE state. Acknowledgment is allowed for transitions to the IDLE state, and has no effect on the state of this object. This Event Enrollment is included in Alarm Enrollment summaries unless filter criteria eliminates it.

`3` **ACK-ALL**. Acknowledgment is REQUIRED FOR ALL notifications specifying transition to the ACTIVE or IDLE states. This is included in the Alarm Enrollment Summaries unless it is eliminated by filter criteria.

NOTE: Acknowledgment is never allowed for event notification specifying a disabled state.

Response/Confirm

The operation-specific data structure described below is used by the server in issuing an AlterEventEnrollment response (**`mp_altee_resp`**). It is received by the client when an AlterEventEnrollment confirm (**`u_mp_altee_conf`**) is received.

```
struct altee_resp_info
  {
  ST_INT16    cur_state_tag;
  ST_INT16    state;
  EVENT_TIME  trans_time;
  SD_END_STRUCT
  };
typedef struct altee_resp_info ALTEE_RESP_INFO;
```

Fields:

`cur_state_tag` This current state tag indicates:

`0` **STATE**. There is a current state defined for this Event Enrollment.

`1` **UNDEFINED**. There is not a current state defined for this Event Enrollment.

`state` This is the current state of the Event Enrollment:

- **`0`** **DISABLED**.
- **`1`** **IDLE**.
- **`2`** **ACTIVE**.
- **`3`** **NO-ACK-A**. Indicates that required acknowledgment for a transition of the Event Condition to the ACTIVE state has not yet been received from the client.

  If the major state is **ACTIVE,** then this is the minor state. Its value of the Time Active Acknowledge (**`aack_time`**) is either UNDEFINED or is earlier than the value of the Time of Last Transition to Active (**`tta_time`**).

  If the major state is **IDLE**, then this is the minor state. Its value of **`tta_time`** is not equal to UNDEFINED, and that the value of **`aack_time`** is either UNDEFINED, or earlier in time than the value of **`tta_time`**.

- **`4`** **NO-ACKED-I**. Indicates that a required acknowledgment for a transition of the Event Condition to the **IDLE** state has not yet been received from the client.

  This minor state is only defined for a major **IDLE** state specifying a value of Alarm Acknowledgment Rule (**`alarm_ack_rule`**) equal to **ACK-ALL**. It indicates that the required acknowledgment for a previous transition to **ACTIVE** state has been received. The value of Time IDLE Acknowledged (**`iack_time`**) is UNDEFINED or earlier than the value of Time of Last Transition to **IDLE** (**`tti_time`**).

- **`5`** **ACKED**. Indicates that all required acknowledgments have been received from the client.

`trans_time` This structure of type **`EVENT_TIME`** contains the time of the last transition of the Event Condition from state to state as described above. See page 2-66 for more information on this structure.

Paired Primitive Interface Functions

mp_altee

Usage: This primitive request function sends an AlterEventEnrollment request PDU to a remote node. It uses the data found in a structure of type **ALTEE_REQ_INFO**, pointed to by **info**. This service is used to request that a VMD replace attributes (Event Condition Transition or Alarm Acknowledgment Rule) of an existing Event Enrollment object.

Function Prototype:

```
MMSREQ_PEND *mp_altee (ST_INT chan,
                       ALTEE_REQ_INFO *info);
```

Parameters:

chan	This is the channel number over which the PDU is to be sent.
info	This pointer to the operation-specific data structure of type **ALTEE_REQ_INFO** contains data specific to the request PDU to be sent.

Return Value:

MMSREQ_PEND *	This pointer to the request control data structure of type **MMSREQ_PEND** is used to send the PDU. In case of an error, the pointer is set to null and **mms_op_err** is written with the error code.

Corresponding User Confirmation Function: **u_mp_altee_conf**

Operation-Specific Data Structure Used: **ALTEE_REQ_INFO**

See page 3-159 for a detailed description of this structure.

u_altee_ind

Usage: This user indication function is called when an AlterEventEnrollment indication is received by a server node. The contents and operation of this function are user-defined, and depend on the application requirements. To conform to the MMS protocol requirements, the application must do one of the following some time after the indication is received:

1) send a positive response using the primitive response function (**`mp_altee_resp`**) after the specified Event Enrollment attributes have been successfully altered, or

2) call the appropriate primitive negative (error) response function (**`mp_err_resp`**). See **Module 11 — MMS-EASE Error Handling** on page 3-343 for an explanation of this function.

See **Volume 1 — Module 1 — Primitive User Confirmation Function Class** for additional information regarding recommended handling of indications.

Function Prototype: `ST_VOID u_altee_ind (MMSREQ_IND *ind);`

Parameters:

`ind` This pointer to the **`MMSREQ_IND`** structure is described in **Volume 1 — Module 2 — Request and Indication Control Data Structures**. This pointer must be used to send the response to this indication. The primary item of interest in this structure is **`ind->req_info_ptr`**. This is a pointer to the operation-specific data structure for an Alter EventEnrollment indication (**`ALTEE_REQ_INFO`**). This pointer will always be valid when **`u_altee_ind`** is called.

Return Value: `ST_VOID` (ignored)

Operation-Specific Data Structure Used: **`ALTEE_REQ_INFO`**

See page 3-159 for a detailed description of this structure.

mp_altee_resp

Usage: This primitive response function sends an AlterEventEnrollment positive response PDU using the data from a structure of type **ALTEE_RESP_INFO**, pointed to by **info**. This function should be called as a response to the **u_altee_ind** function being called (an AlterEvent Enrollment indication is received), and after the specified Event Enrollment attributes have been successfully altered.

Function Prototype:

```
ST_RET mp_altee_resp (MMSREQ_IND *ind,
                      ALTEE_RESP_INFO *info);
```

Parameters:

`ind` This pointer to the indication control structure of type **MMSREQ_IND** is received in the **u_altee_ind** function.

`info` This pointer to an Operation-Specific data structure of type **ALTEE_RESP_INFO** contains information specific to the response PDU being sent.

Return Value: `ST_RET` **SD_SUCCESS**. No Error

`<> 0` Error Code

Corresponding User Indication Function: **u_altee_ind**

Operation-Specific Data Structure Used: **ALTEE_RESP_INFO**

See page 3-160 for a detailed description of this structure.

u_mp_altee_conf

Usage: This primitive user confirmation function is called when a confirm to an AlterEventEnrollment request (**mp_altee**) is received. **resp_err** contains a value indicating whether an error occurred. **resp_info_ptr** may contain data that the remote node sent back in the confirmation or it may contain error information if an error occurred. If no error has occurred (**req->resp_err = CNF_RESP_OK**), AlterEventEnrollment information is available to the application using the **req->resp_info_ptr** pointer. See **Volume 1 — Module 1 — Primitive User Confirmation Function Class** for specific information on the recommended handling of confirms.

Note that the function called for this event may be changed if desired; however, **u_mp_altee_conf** is the standard default function.

Function Prototype:

```
ST_VOID u_mp_altee_conf (MMSREQ_PEND *req);
```

Parameters:

`req` This pointer to the **MMSREQ_PEND** structure is returned from the original **mp_altee** request. See **Volume 1 — Module 2 — Request and Indication Control Data Structures** for information on the structure.

For a positive response (**req->resp_err = CNF_RESP_OK**), the request control structure member **resp_info_ptr** is a pointer to the operation-specific data structure (**ALTEE_RESP_INFO**) for an AlterEventEnrollment function.

For a negative response, or other errors, see **Module 11 — MMS-EASE Error Handling** starting on page 3-339 for more information.

Return Value: `ST_VOID` (ignored)

Operation-Specific Data Structure Used: **ALTEE_RESP_INFO**

See page 3-160 for a detailed description of this structure.

28. EventNotification Service

This service provides the VMD with the ability to notify an enrolled client of the occurrence of a state transition associated with an Event Condition object. This is an unconfirmed service.

Primitive Level EventNotification Operations

The following section contains information on how to use the paired primitive interface for the EventNotification service. It covers available data structures used by the PPI, and the two primitive level functions that together make up the EventNotification service. See **Volume 1 — Module 2 — General Sequence of Events — Unconfirmed Services** for a detailed explanation of the sequence of events that occur within the MMS-EASE environment during unconfirmed services.

- The EventNotification service consists of the paired primitive functions of **`mp_evnot`**, and **`u_evnot_ind`**.

Data Structures

Request/Indication

The operation-specific structure described below is used by the client in issuing an EventNotification request (**`mp_evnot`**). It is received by the server when an EventNotification indication (**`u_evnot_ind`**) is received.

```
struct evnot_req_info
  {
  OBJECT_NAME evenroll_name;
  OBJECT_NAME evcon_name;
  ST_UCHAR    severity;
  ST_BOOLEAN  cur_state_pres;
  ST_INT16    cur_state;
  EVENT_TIME  trans_time;
  ST_UCHAR    not_lost;
  ST_BOOLEAN  alarm_ack_rule_pres;
  ST_INT16    alarm_ack_rule;
  ST_BOOLEAN  evact_result_pres;
  OBJECT_NAME evact_name;
  ST_INT16    evact_result_tag;
  ST_INT      conf_serv_resp_len;
  ST_UCHAR    *conf_serv_resp;
  ST_BOOLEAN  cs_rdetail_pres;
  ST_INT      cs_rdetail_len;
  ST_UCHAR    *cs_rdetail;
  ST_BOOLEAN  mod_pos_pres;
  ST_UINT32   mod_pos;
  ERR_INFO    *serv_err;
  SD_END_STRUCT
  };
typedef struct evnot_req_info EVNOT_REQ_INFO;
```

Fields:

`evenroll_name` This structure of type **`OBJECT_NAME`** contains the name of the event enrollment for which this EventNotification is intended. See **Volume 1 — Module 2 — MMS Object Name Structure** for more information on this structure.

`evcon_name` This contains the name of the Event Condition for which the EventNotification is being issued.

`severity` This indicates the severity of the Event Condition:

`0`	Most severe
:	:
`255`	Least severe

`cur_state_pres` **`SD_FALSE`**. Do NOT include **`cur_state`** in the PDU.

`SD_TRUE`. Include **`cur_state`** in the PDU.

`cur_state` This indicates the current state of the event condition:

`0`	DISABLED
`1`	IDLE
`2`	ACTIVE

`trans_time` This structure of type **`EVENT_TIME`** contains the time of the transition of the event condition as described above. See page 3-66 for an explanation of this structure.

`not_lost` **`SD_FALSE`**. Previous event notifications for previous transitions of the event condition corresponding to this event enrollment have NOT been lost. This is the default.

`SD_TRUE`. One or more previous event notifications for previous transitions of the event condition corresponding to this event enrollment have been lost. This can occur when a previous event notification should have been sent, but was not. This is due to resource limitations at the VMD, or because an association could not be established.

`alarm_ack_rule_pres` **`SD_FALSE`**. Do NOT include **`alarm_ack_rule`** in the PDU.

`SD_TRUE`. Include **`alarm_ack_rule`** in the PDU.

`alarm_ack_rule` The alarm acknowledgment rule specifies the conditions under which the AcknowledgeEventNotification (**`mp_ackevnot`**) service must be used:

- **`0`** **NONE**. Acknowledgment is not supported.
- **`1`** **SIMPLE**. Acknowledgment is allowed, but optional for transitions to **ACTIVE** or **IDLE**.
- **`2`** **ACK_ACTIVE**. Acknowledgment is mandatory for transitions to **ACTIVE**, and optional for transitions to **IDLE**.
- **`3`** **ACK_ALL**. Acknowledgment is mandatory for all transitions to **ACTIVE** or **IDLE**.

`evact_result_pres` **`SD_FALSE`**. Do NOT include **`evact_result_tag`** and **`evact_result`** in the PDU.

`SD_TRUE`. Include **`evact_result_tag`** and **`evact_result`** in the PDU.

`evact_name` This structure of type **`OBJECT_NAME`** contains the name of the Event Action corresponding to the event condition. Use only if **`evact_result_pres != SD_FALSE`**.

`evact_result_tag` This contains the event action result tag. Use only if **`evact_result_pres != SD_FALSE`**. It indicates whether the confirmed service request specified by the event action was acted upon:

- **`0`** **SUCCESS.** The confirmed service request was acted upon and resulted in a response (positive or negative).
- **`1`** **FAILURE**. The confirmed service request specified by the event action was never acted upon.

`conf_serv_resp_len` If **`evact_result_tag = 0`**, this is length, in bytes, of the data pointed to by **`conf_serv_resp`**.

`conf_serv_resp` This is a pointer to the confirmed service response. This is the result (+) or result (-) of the confirmed service response specified for the event action, and only is included if **`evact_result_tag = 0`** (success).

`cs_rdetail_pres` **`SD_FALSE`**. Do NOT include **`cs_rdetail`** in the PDU.

`SD_TRUE`. Include **`cs_rdetail`** in the PDU.

`cs_rdetail_len` If **`evact_result_tag = 0`**, this is the length, in bytes, of the data pointed to by **`cs_rdetail`**.

`cs_rdetail` This is a pointer to the Companion Standard request detail, and only is included if **`evact_result_tag = 0`**.

`mod_pos_pres` **`SD_FALSE`**. The **`mod_pos`** member is not active.

`SD_TRUE`. The **`mod_pos`** member is present.

`mod_pos` If **`evact_result_tag != 0`**, this indicates the position of the modifier to which the error pertains.

`serv_err` If **`evact_result_tag != 0`**, this structure of type **`ERR_INFO`** contains a pointer to the service error.

Paired Primitive Interface Functions

mp_evnot

Usage: This primitive request function sends an EventNotification request PDU to a remote node. It uses the data found in a structure of type **EVNOT_REQ_INFO** pointed to by **info**. This service is used to inform a remote node with an event enrollment of the occurrence of a transition of an Event Condition. There is no confirmation to this request since it is an unconfirmed service request. However, it is the responsibility of the remote node with the event enrollment to later acknowledge receipt of the EventNotification. This is done by sending an AcknowledgeEventNotification request. See page 3-174 for more information on **mp_ackevnot**.

Function Prototype: `ST_RET mp_evnot (ST_INT chan, EVNOT_REQ_INFO *info);`

Parameters:

`chan`	This is the channel number over which the PDU is to be sent.
`info`	This pointer to the operation-specific data structure of type **EVNOT_REQ_INFO** contains data specific to the request PDU to be sent.

Return Value: `ST_RET` **SD_SUCCESS**. No Error

`<> 0` Error Code

Corresponding User Confirmation Function: NONE

Operation-Specific Data Structure Used: **EVNOT_REQ_INFO**

See page 3-167 for a detailed description of this structure.

u_evnot_ind

Usage: This user indication function is called when an EventNotification indication is received by a server node. The contents and operation of this function are user-defined, and depend on the application requirements.

See **Volume 1 — Module 1 — User Indication Function Class** for additional information regarding the recommended handling of indications.

Function Prototype: `ST_VOID u_evnot_ind (MMSREQ_IND *ind);`

Parameters:

`ind` This pointer to the **`MMSREQ_IND`** structure is described in **Volume 1 — Module 2 — Request and Indication Control Data Structures**. This pointer must be used to send the response to this indication. The primary item of interest in this structure is **`ind->req_info_ptr`**. This is a pointer to the operation-specific data structure for an EventNotification indication (**`EVNOT_REQ_INFO`**). This pointer will always be valid when **`u_evnot_ind`** is called.

Return Value: `ST_VOID` (ignored)

Operation-Specific Data Structure Used: **`EVNOT_REQ_INFO`**

See page 3-167 for a detailed description of this structure.

29. AcknowledgeEventNotification Service

This service is used by the client to notify the VMD that an EventNotification has been acknowledged as received from the VMD.

Primitive Level AcknowledgeEventNotification Operations

The following section contains information on how to use the paired primitive interface for the AcknowledgeEventNotification service. It covers available data structures used by the PPI, and the four primitive level functions that together make up the AcknowledgeEventNotification service. See **Volume 1 — Module 2 — General Sequence of Events — Confirmed Services** for a detailed explanation of the sequence of events that occur within the MMS-EASE environment during confirmed services.

- The AcknowledgeEventNotification service consists of the paired primitive functions of **`mp_ackevnot`**, **`u_ackevnot_ind`**, **`mp_ackevnot_resp`**, and **`u_mp_ackevnot_conf`**.

Data Structures

Request/Indication

The operation-specific structure described below is used by the client in issuing an AcknowledgeEventNotification request (**`mp_ackevnot`**). It is received by the server when an AcknowledgeEventNotification indication (**`u_ackevnot_ind`**) is received.

```
struct ackevnot_req_info
  {
  OBJECT_NAME evenroll_name;
  ST_INT16    ack_state;
  EVENT_TIME  evtime;
  ST_BOOLEAN  ackec_name_pres;
  OBJECT_NAME ackec_name;
  };
typedef struct ackevnot_req_info ACKEVNOT_REQ_INFO;
```

Fields:

`evenroll_name` This structure of type **`OBJECT_NAME`** contains the MMS object name of the Event Enrollment being acknowledged. See **Volume 1 — Module 2 — MMS Object Name Structure** for an explanation of this structure.

`ack_state` This contains the state of the acknowledgment. This should be equal to the current state of the EventNotification being acknowledged:

- **`0`** DISABLE
- **`1`** IDLE
- **`2`** ACTIVE

`evtime` This structure of type **`EVENT_TIME`** contains the time of the acknowledged transition. It should be equal to the transition time of the EventNotification being acknowledged. See page 3-66 for an explanation of the **`event_time`** structure.

`ackec_name_pres` **`SD_FALSE`**. Do NOT include **`ackec_name`** in the PDU.

`SD_TRUE`. Include **`ackec_name`** in the PDU.

`ackec_name` This structure of type **`OBJECT_NAME`** contains the name of the AcknowledgeEventCondition. Specifying an AcknowledgeEventCondition name is optional.

Paired Primitive Interface Functions

mp_ackevnot

Usage: This primitive request function sends an AcknowledgeEventNotification request PDU to a remote node. It uses the data found in a structure of type **ACKEVNOT_REQ_INFO**, pointed to by **info**. This service is used to tell the remote node that the local node has acknowledged an EventNotification previously sent by the remote node.

Function Prototype:

```
MMSREQ_PEND *mp_ackevnot (ST_INT chan,
                          ACKEVNOT_REQ_INFO *info);
```

Parameters:

chan	This is the channel number over which the PDU is to be sent.
info	This pointer to the operation-specific data structure of type **ACKEVNOT_REQ_INFO** contains data specific to the request PDU to be sent.

Return Value:

MMSREQ_PEND *	This pointer to the request control data structure of type **MMSREQ_PEND** is used to send the PDU. In case of an error, the pointer is set to null and **mms_op_err** is written with the error code.

Corresponding User Confirmation Function: **u_mp_ackevnot_conf**

Operation-Specific Data Structure Used: **ACKEVNOT_REQ_INFO**

See page 3-173 for a detailed description of this structure.

u_ackevnot_ind

Usage: This user indication function is called when an AcknowledgeEventNotification indication is received by a server node. The contents and operation of this function are user-defined, and depend on the application requirements. To conform to the MMS protocol requirements, the application must do one of the following some time after the indication is received:

1) send a positive response using the primitive response function (**`mp_ackevnot_resp`**) after the specified EventNotification has been acknowledged, or

2) call the appropriate primitive negative (error) response function (**`mp_err_resp`**). See **Module 11 — MMS-EASE Error Handling** on page 3-343 for an explanation of this function.

See **Volume 1 — Module 1 — User Indication Function Class** for additional information regarding the recommended handling of indications.

Function Prototype: `ST_VOID u_ackevnot_ind (MMSREQ_IND *ind);`

Parameters:

`ind` This pointer to the **`MMSREQ_IND`** structure is described in **Volume 1 — Module 2 — Request and Indication Control Data Structures**. This pointer must be used to send the response to this indication. The primary item of interest in this structure is **`ind->req_info_ptr`**. This is a pointer to the operation-specific data structure for an AcknowledgeEventNotification indication (**`ACKEVNOT_REQ_INFO`**). This pointer will always be valid when **`u_ackevnot_ind`** is called.

Return Value: `ST_VOID` (ignored)

Operation-Specific Data Structure Used: **`ACKEVNOT_REQ_INFO`**

See page 3-173 for a detailed description of this structure.

mp_ackevnot_resp

Usage: This primitive response function sends an AcknowledgeEventNotification positive response PDU. This function should be called after the **u_ackevnot_ind** function is called (an AcknowledgeEventNotification indication is received), and after the specified EventNotification has been acknowledged.

Function Prototype: `ST_RET mp_ackevnot_resp (MMSREQ_IND *info);`

Parameters:

`info` This pointer to the indication control structure of type **MMSREQ_IND** is received in the **u_ackevnot_ind** function.

Return Value: `ST_RET` **SD_SUCCESS**. No Error.

`<> 0` Error Code.

Corresponding User Indication Function: **u_ackevnot_ind**

Operation-Specific Data Structure Used: NONE

u_mp_ackevnot_conf

Usage: This primitive user confirmation function is called when a confirm to an AcknowledgeEventNotification request (**mp_ackevnot**) is received. **resp_err** contains a value indicating whether an error occurred. **resp_info_ptr** may contain data that the remote node sent back in the confirmation or it may contain error information if an error occurred. If no error has occurred (**req->resp_err = CNF_RESP_OK**), AcknowledgeEventNotification information is available to the application using the **req->resp_info_ptr** pointer. See **Volume 1 — Module 1 — Primitive User Confirmation Function Class** for specific information on the recommended handling of confirms.

Note that the function called for this event may be changed if desired; however, **u_mp_ackevnot_conf** is the standard default function.

Function Prototype:

```
ST_VOID u_mp_ackevnot_conf (MMSREQ_PEND *req);
```

Parameters:

`req` This pointer to the **MMSREQ_PEND** structure is returned from the original **mp_ackevnot** request. See **Volume 1 — Module 2 — Request and Indication Control Data Structures** for information on the structure.

For a positive response (**req->resp_err = CNF_RESP_OK**), the request control structure member **resp_info_pres** indicates that there is no confirm data for this function.

For a negative response, or other errors, see **Module 11 — MMS-EASE Error Handling** starting on page 3-339 for more information.

Return Value: `ST_VOID` (ignored)

Operation Specific Data Structure Used: NONE

30. GetAlarmSummary Service

This service provides the client with the ability to request summary information from the VMD regarding the current status of Event Condition objects for which the Alarm Summary Reports attribute is true.

Primitive Level GetAlarmSummary Operations

The following section contains information on how to use the paired primitive interface for the GetAlarmSummary service. It covers available data structures used by the PPI, and the four primitive level functions that together make up the GetAlarmSummary service. See **Volume 1 — Module 2 — General Sequence of Events — Confirmed Services** for a detailed explanation of the sequence of events that occur within the MMS-EASE environment during confirmed services.

- The GetAlarmSummary service consists of the paired primitive functions of **`mp_getas`**, **`u_getas_ind`**, **`mp_getas_resp`**, and **`u_mp_getas_conf`**.

Data Structures

Request/Indication

The operation-specific structure described below is used by the client in issuing a GetAlarmSummary request (**`mp_getas`**). It is received by the server when a GetAlarmSummary indication (**`u_getas_ind`**) is received.

```
struct getas_req_info
  {
  ST_BOOLEAN  enroll_only;
  ST_BOOLEAN  act_alarms_only;
  ST_INT16    ack_filter;
  ST_UCHAR    most_sev_filter;
  ST_UCHAR    least_sev_filter;
  ST_BOOLEAN  ca_pres;
  OBJECT_NAME ca_name;
typedef struct getas_req_info GETAS_REQ_INFO;
```

Fields:

`enroll_only` — **`SD_FALSE`**. Report all Event Conditions.

`SD_TRUE`. Report only Event Conditions for which you are enrolled.

`act_alarms_only` — **`SD_FALSE`**. Report all applicable Event Conditions

`SD_TRUE`. Report only Event Conditions that are ACTIVE.

`ack_filter` — This contains the acknowledgment filter:

- **`0`** **NOT-ACKED**(default). Report only unacknowledged Event Conditions.
- **`1`** **ACKED**. Report only acknowledged Event Conditions.
- **`2`** **ALL**. Report all Event conditions.

`most_sev_filter` — This is the minimum value of severity for the Event Condition to be returned.

`least_sev_filter` — This is the maximum value of severity for the Event Condition to be returned.

`ca_pres` — **`SD_FALSE`**. Do NOT include **`ca_name`** in the PDU.

`SD_TRUE`. Include **`ca_name`** in the PDU.

ca_name — This structure of type **OBJECT_NAME** contains the Event Condition Name to continue after. It allows making multiple requests to read the entire alarm summary if it could not be sent in a single response. See **Volume 1 — Module 2 — MMS Object Name Structure** for more information on this structure.

Response/Confirm

The operation-specific data structure described below is used by the server in issuing a GetAlarmSummary response (**mp_getas_resp**). It is received by the client when a GetAlarmSummary confirm (**u_mp_getas_conf**) is received.

```
struct getas_resp_info
  {
  ST_BOOLEAN    more_follows;
  ST_INT        num_of_alarm_sum;
/*ALARM_SUMMARY alarm_sum [num_of_alarm_sum];                    */
  SD_END_STRUCT
  };
typedef struct getas_resp_info GETAS_RESP_INFO;
```

Fields:

more_follows — **SD_FALSE**. No more information follows.

SD_TRUE. There are more alarm summaries available that are not included in this response.

num_of_alarm_sum — This contains the number of alarm summaries included in this response. It is equal to the number of elements in the **alarm_sum** array of structures.

alarm_sum — This array of structures of type **ALARM_SUMMARY** contains the list of alarm summaries to be included in this response. See below for a detailed description of this structure.

NOTE: When allocating a data structure of type **GETAS_RESP_INFO**, enough memory must be allocated to hold the information for the actual alarm summaries. The following C language statement can be used:

```
info = (GETAS_RESP_INFO *) chk_malloc(sizeof (GETAS_RESP_INFO) +
             (num_of_alarm_sum * sizeof(ALARM_SUMMARY)));
```

```
struct alarm_summary
  {
  OBJECT_NAME evcon_name;
  ST_UCHAR    severity;
  ST_INT16    cur_state;
  ST_INT16    unack_state;
  ST_BOOLEAN  addl_detail_pres;
  ST_INT      addl_detail_len;
  ST_UCHAR    *addl_detail;
  ST_BOOLEAN  tta_time_pres;
  EVENT_TIME  tta_time;
  ST_BOOLEAN  tti_time_pres;
  EVENT_TIME  tti_time;
  SD_END_STRUCT
  };
typedef struct alarm_summary ALARM_SUMMARY;
```

Fields:

`evcon_name` This structure of type **`OBJECT_NAME`** contains the name of the Event Condition concerning this Alarm Summary.

`severity` This is a value indicating the Event Condition's severity with 0 = most severe, and 255 = least severe.

`cur_state` This indicates the current state of the Event Condition:

- **`0`** Event Condition Disabled
- **`1`** Event Condition Idle
- **`2`** Event Condition Active

`unack_state` This is a value indicating the state(s) of the Event Condition for which at least one Event Enrollment has not been acknowledged:

- **`0`** **NONE**
- **`1`** **ACTIVE**
- **`2`** **IDLE**
- **`3`** **BOTH** (The ACTIVE & IDLE transitions are both unacked)

`addl_detail_pres` **`SD_FALSE`**. Do NOT include **`addl_detail`** in the PDU.

`SD_TRUE`. Include **`addl_detail`** in the PDU.

`addl_detail_len` This is the length, in bytes, of the data pointed to by **`addl_detail`**.

`addl_detail` This is the Additional Detail. The content and form of this data must be valid ASN.1.

`tta_time_pres` **`SD_FALSE`**. Do NOT include **`tta_time`** in the PDU.

`SD_TRUE`. Include **`tta_time`** in the PDU.

`tta_time` This structure of type **`EVENT_TIME`** contains the time of the last transition to ACTIVE[7].

`tti_time_pres` **`SD_FALSE`**. Do NOT include **`tti_time`** in the PDU.

`SD_TRUE`. Include **`tti_time`** in the PDU.

`tti_time` This structure of type **`EVENT_TIME`** contains the time of the last transition to IDLE[8].

NOTE: See page 3-66 for a detailed description of the **`EVENT_TIME`** structure. See **Volume 1 — Module 2 — MMS Object Name Structure** for more information on this structure.

[7] Include these fields only if there has been a corresponding transition **and** the event condition's alarm summary reports attribute is TRUE, **or** if the event condition is referenced by an event enrollment whose alarm acknowledgment rule is not NONE.

Paired Primitive Interface Functions

mp_getas

Usage: This primitive request function sends a GetAlarmSummary request PDU to a remote node. It uses the data found in a structure of type **GETAS_REQ_INFO** pointed to by **info**. This service is used to obtain summary information about the status of various Event Conditions at a remote VMD.

Function Prototype:

```
MMSREQ_PEND *mp_getas (ST_INT chan,
                       GETAS_REQ_INFO *info);
```

Parameters:

chan	This is the channel number over which the PDU is to be sent.
info	This pointer to the operation-specific data structure of type **GETAS_REQ_INFO** contains data specific to the request PDU to be sent.

Return Value:

MMSREQ_PEND *	This pointer to the request control data structure of type **MMSREQ_PEND** is used to send the PDU. In case of an error, the pointer is set to null and **mms_op_err** is written with the error code.

Corresponding User Confirmation Function: **u_mp_getas_conf**

Operation-Specific Data Structure Used: **GETAS_REQ_INFO**

See page 3-179 for a detailed description of this structure.

u_getas_ind

Usage: This user indication function is called when a GetAlarmSummary indication is received by a server node. The contents and operation of this function are user-defined, and depend on the application requirements. To conform to the MMS protocol requirements, the application must do one of the following some time after the indication is received:

1) send a positive response using the primitive response function (**`mp_getas_resp`**) after the specified Alarm Summary information has been successfully obtained, or

2) call the appropriate primitive negative (error) response function (**`mp_err_resp`**). See **Module 11 — MMS-EASE Error Handling** on page 343 for an explanation of this function.

See **Volume 1 — Module 1 — User Indication Function Class** for additional information regarding recommended handling of indications.

Function Prototype:

```
ST_VOID u_getas_ind  (MMSREQ_IND *ind);
```

Parameters:

`ind` This pointer to the **`MMSREQ_IND`** structure is described in **Volume 1 — Module 2 — Request and Indication Control Data Structures**. This pointer must be used to send the response to this indication. The primary item of interest in this structure is **`ind->req_info_ptr`**. This is a pointer to the operation-specific data structure for a GetAlarmSummary indication (**`GETAS_REQ_INFO`**). This pointer will always be valid when **`u_getas_ind`** is called.

Return Value: `ST_VOID` (ignored)

Operation-Specific Data Structure Used: `GETAS_REQ_INFO`

See page 3-179 for a detailed description of this structure.

mp_getas_resp

Usage: This primitive response function sends a GetAlarmSummary positive response PDU using the data from a structure of type **GETAS_RESP_INFO**, pointed to by **info**. This function should be called as a response to the **u_getas_ind** function being called (a GetAlarmSummary indication is received), and after the specified Alarm Summary information has been successfully obtained.

Function Prototype:

```
ST_RET mp_getas_resp (MMSREQ_IND *ind,
                      GETAS_RESP_INFO *info);
```

Parameters:

`ind` This pointer to the indication control structure of type **MMSREQ_IND** is received in the **u_getas_ind** function.

`info` This pointer to an Operation-Specific data structure of type **GETAS_RESP_INFO** contains information specific to the response PDU to be sent.

Return Value: `ST_RET` **SD_SUCCESS**. No Error

`<> 0` Error Code

Corresponding User Indication Function: **u_getas_ind**

Operation-Specific Data Structure Used: **GETAS_RESP_INFO**

See page 3-180 for a detailed description of this structure.

u_mp_getas_conf

Usage: This primitive user confirmation function is called when a confirm to a GetAlarmSummary request (**mp_getas**) is received. **resp_err** contains a value indicating whether an error occurred. **resp_info_ptr** may contain data that the remote node sent back in the confirmation or it may contain error information if an error occurred. If no error has occurred (**req->resp_err = CNF_RESP_OK**), GetAlarmSummary information is available to the application using the **req->resp_info_ptr** pointer. See **Volume 1 — Module 1 — Primitive User Confirmation Function Class** for specific information on the recommended handling of confirms.

Note that the function called for this event may be changed if desired; however, **u_mp_getas_conf** is the standard default function.

Function Prototype:

```
ST_VOID u_mp_getas_conf (MMSREQ_PEND *req);
```

Parameters:

`req` This pointer to the **MMSREQ_PEND** structure is returned from the original **mp_getas** request. See **Volume 1 — Module 2 — Request and Indication Control Data Structures** for information on the structure.

For a positive response (**req->resp_err = CNF_RESP_OK**), the request control structure member **resp_info_ptr** is a pointer to the operation-specific data structure (**GETAS_RESP_INFO**) for the GetAlarmSummary function.

For a negative response, or other errors, see **Module 11 — MMS-EASE Error Handling** starting on page 3-339 for more information.

Return Value: `ST_VOID` (ignored)

Operation-Specific Data Structure Used: **GETAS_RESP_INFO**

See page 3-180 for a detailed description of this structure.

31. GetAlarmEnrollmentSummary Service

This service is used to request summary information from the VMD. This information is regarding the current status of Event Enrollment objects where the value of the Alarm Acknowledgment Rule attribute is not equal to NONE (**`alarm_ack_rule != 0`**).

Primitive Level GetAlarmEnrollmentSummary Operations

The following section contains information on how to use the paired primitive interface for the GetAlarmEnrollmentSummary service. It covers available data structures used by the PPI, and the four primitive level functions that together make up the GetAlarmEnrollmentSummary service. See **Volume 1 — Module 2 — General Sequence of Events — Confirmed Services** for a detailed explanation of the sequence of events that occur within the MMS-EASE environment during confirmed services.

- The GetAlarmEnrollmentSummary service consists of the paired primitive functions of **`mp_getaes`**, **`u_getaes_ind`**, **`mp_getaes_resp`**, and **`u_mp_getaes_conf`**.

Data Structures

Request/Indication

The operation-specific structure described below is used by the client in issuing a GetAlarmEnrollmentSummary request (**`mp_getaes`**). It is received by the server when a GetAlarmEnrollmentSummary indication (**`u_getaes_ind`**) is received.

```
struct getaes_req_info
  {
  ST_BOOLEAN  enroll_only;
  ST_BOOLEAN  act_alarms_only;
  ST_INT16    ack_filter;
  ST_UCHAR    most_sev_filter;
  ST_UCHAR    least_sev_filter;
  ST_BOOLEAN  ca_name_pres;
  OBJECT_NAME ca_name;
  };
typedef struct getaes_req_info GETAES_REQ_INFO;
```

Fields:

`enroll_only` **`SD_FALSE`**. Report all Event Enrollments.

`SD_TRUE`. Report only Event Enrollments for which you are enrolled.

`act_alarms_only` **`SD_FALSE`**. Report all applicable Event Enrollments.

`SD_TRUE`. Report only Event Enrollments that are ACTIVE.

`ack_filter` This is the acknowledgment filter:

`0` **NOT-ACKED**. Report only unacknowledged Event Enrollments.

`1` **ACKED**. Report only acknowledged Event Enrollments.

`2` **ALL**. Report all Event Enrollments.

most_sev_filter	This is the minimum value of severity for the Event Enrollments to be returned.
least_sev_filter	This is the maximum value of severity for the Event Enrollments to be returned.
ca_name_pres	**SD_FALSE**. Do NOT include **ca_name** in the PDU. **SD_TRUE**. Include **ca_name** in the PDU.
ca_name	This structure of type **OBJECT_NAME** contains the Event Enrollment Name to continue after. It allows making multiple requests to read the entire alarm summary if it could not be sent in a single response. See **Volume 1 — Module 2 — MMS Object Name Structure** for more information on this structure.

Response/Confirm

The operation-specific data structure described below is used by the server in issuing a GetAlarmEnrollment-Summary response (**mp_getaes_resp**). It is received by the client when a GetAlarmEnrollmentSummary confirm (**u_mp_getaes_conf**) is received.

```
struct getaes_resp_info
  {
  ST_BOOLEAN           more_follows;
  ST_INT               num_of_alarm_esum;
/*ALARM_ENROLL_SUMMARY alarm_enroll_sum [num_of_alarm_esum];*/
  SD_END_STRUCT
  };
typedef struct getaes_resp_info  GETAES_RESP_INFO;
```

Fields:

more_follows	**SD_FALSE**. No more information follows. **SD_TRUE**. There are more alarm summaries available not included in this response.
num_of_alarm_esum	This indicates the number of alarm enrollment summaries included in this response. This is equal to the number of elements in the **alarm_enroll_sum** array of structures.
alarm_enroll_sum	This array of structures of type **ALARM_ENROLL_SUMMARY** contains the list of alarm enrollment summaries to be included in this response. Please refer to the next page for a detailed description of this structure.

NOTE: When allocating a data structure of type **GETAES_RESP_INFO**, enough memory must be allocated to hold the information for the actual alarm summaries. The following C language statement can be used:

```
info = (GETAES_RESP_INFO *) chk_malloc(sizeof(GETAES_RESP_INFO) +
             (num_of_alarm_esum * sizeof(ALARM_SUMMARY)));
```

```
struct alarm_enroll_summary
  {
  OBJECT_NAME evenroll_name;
  ST_BOOLEAN  client_app_pres;
  ST_INT      client_app_len;
  ST_UCHAR    *client_app;
  ST_UCHAR    severity;
  ST_INT16    cur_state;
  ST_BOOLEAN  addl_detail_pres;
  ST_INT      addl_detail_len;
  ST_UCHAR    *addl_detail;
  ST_BOOLEAN  not_lost;
  ST_INT16    alarm_ack_rule;
  ST_BOOLEAN  ee_state_pres;
  ST_INT16    ee_state;
  ST_BOOLEAN  tta_time_pres;
  EVENT_TIME  tta_time;
  ST_BOOLEAN  aack_time_pres;
  EVENT_TIME  aack_time;
  ST_BOOLEAN  tti_time_pres;
  EVENT_TIME  tti_time;
  ST_BOOLEAN  iack_time_pres;
  EVENT_TIME  iack_time;
  SD_END_STRUCT
  };
typedef struct alarm_enroll_summary ALARM_ENROLL_SUMMARY;
```

Fields:

`evenroll_name` This structure of type **`OBJECT_NAME`** contains the name of the event enrollment for which this alarm enrollment summary is intended. See **Volume 1 — Module 2 — MMS Object Name Structure** for more information on this structure.

`client_app_pres` **`SD_FALSE`**. Do NOT include **`client_app`** in the PDU.

`SD_TRUE`. Include **`client_app`** in the PDU.

`client_app_len` This is the length, in bytes, of the data pointed to in **`client_app`**.

`client_app` This is a pointer to the client application reference containing the identification of the enrolled client application.

`severity` This is a value indicating the alarm enrollment summary's severity with 0 = most severe, and 255 = least severe.

`curr_state` This is the current state of the corresponding Event Condition:

- **`0`** Event Condition is DISABLED.
- **`1`** Event Condition is IDLE.
- **`2`** Event Condition is ACTIVE.

`addl_detail_pres` **`SD_FALSE`**. Do NOT include **`addl_detail`** in the PDU.

`SD_TRUE`. Include **`addl_detail`** in the PDU.

`addl_detail_len` This is the length, in bytes, of the data pointed to in **`addl_detail`**.

`addl_detail` This contains the Additional Detail. The content and form of this data must be valid ASN.1.

`not_lost` **SD_FALSE**. There is no difficulty. Event Notifications are being sent. This is the default.

SD_TRUE. During periods of time when the invocation of the Event Notification service associated (with the Event Enrollment) has been suspended because of some lack of resources. Notifications can be lost because of a lack of CPU resources (when priority is low), either for generating the Event Notification or for carrying out the Event Action.

NOTE: During intervals in which Event Notifications are suspended due to lack of resources, transitions of Event Conditions do not generate additional Event Notifications.

`alarm_ack_rule` This only exists for Event Enrollments that reference a monitored Event Condition. It indicates the level of acknowledgment required for Event Notification instances generated as a result of this Alarm Enrollment Summary. This value also determines whether this Event Enrollment should be included in Alarm Enrollment Summaries. It contains one of the following values:

0 **NONE**. This Event Enrollment is NOT included in Alarm Enrollment Summaries. Acknowledgments to Event Notifications in response to these Event Enrollments are allowed, and have no effect on the state of this object.

1 **SIMPLE**. Acknowledgment is allowed but not required for Event Notifications specifying a transition to ACTIVE. This has effect on the state of the Event Enrollment object. Acknowledgments to Event Notification specifying a transition to the IDLE state is allowed, and has no effect on the state of this object.

2 **ACK-ACTIVE**. Acknowledgment is REQUIRED for notifications specifying transition to the ACTIVE state. Acknowledgment is allowed for transitions to the IDLE state, and has no effect on the state of this object. This Event Enrollment is included in Alarm Enrollment summaries unless it is eliminated by filter criteria.

3 **ACK-ALL**. Acknowledgment is REQUIRED FOR ALL notifications specifying transition to the ACTIVE or IDLE states. This is included in the Alarm Enrollment Summaries unless it is eliminated by filter criteria.

NOTE: Acknowledgment is never allowed for event notification specifying a disabled state.

`ee_state_pres` **SD_FALSE**. Do NOT include **ee_state** in the PDU.

SD_TRUE. Include **ee_state** in the PDU.

`ee_state` This is the current state of the Event Enrollment. At any instance, an Event Enrollment has a major state. This is either NON-EXISTENT or equal to the state of the Event Condition object referenced. Depending on the value of **alarm_ack_rule**, there also may be an implied minor state. This minor state indicates whether the Event Enrollment is waiting for a required acknowledgment from the client for an Event Condition which transitions from ACTIVE to IDLE state. The meanings are as follows:

0 **DISABLED**.

1 **IDLE**.

2 **ACTIVE**.

3 **NO-ACK-A**. Indicates that required acknowledgment for a transition of the Event Condition to the ACTIVE state has not yet been received from the client.

If the major state is **ACTIVE**, then this is the minor state. Its value of the Time Active Acknowledge (**aack_time**) is either UNDEFINED or is earlier than the value of the Time of Last Transition to Active (**tta_time**).

If the major state is **IDLE**, then this is the minor state. Its value of **tta_time** is not equal to UNDEFINED, and that the value of **aack_time** is either UNDEFINED, or earlier in time than the value of **tta_time**.

4 **NO-ACKED-I**. Indicates that a required acknowledgment for a transition of the Event Condition to the **IDLE** state has not yet been received from the client.

This minor state is only defined for a major **IDLE** state specifying a value of Alarm Acknowledgment Rule (**alarm_ack_rule**) equal to **ACK-ALL**. It indicates that the required acknowledgment for a previous transition to **ACTIVE** state has been received. The value of Time IDLE Acknowledged (**iack_time**) is UNDEFINED or earlier than the value of Time of Last Transition to **IDLE** (**tti_time**).

5 **ACKED**. Indicates that all required acknowledgments have been received from the client.

tta_time_pres **SD_FALSE**. Do NOT include **tta_time** in the PDU.

SD_TRUE. Include **tta_time** in the PDU.

tta_time This structure of type **EVENT_TIME** indicates the Time of the Last Transition to Active. See page 3-66 for information on this structure.

aack_time_pres **SD_FALSE**. Do NOT include **aack_time** in the PDU.

SD_TRUE. Include **aack_time** in the PDU.

aack_time This structure of type **EVENT_TIME** indicates the Active Acknowledgment Time. This only exists for notification event enrollments which reference a monitored Event Condition. This is only significant when the value of the Alarm Acknowledgment Rule (**alarm_ack_rule**) is NOT equal to NONE.

It records the time (date and time or Time Sequence Identifier). This is the time at which an acknowledgment for the most recently detected transition of the Event Condition to the ACTIVE state was received from the enrolled client. If this acknowledgment was not received, this value equals UNDEFINED.

tti_time_pres **SD_FALSE**. Do NOT include **tti_time** in the PDU.

SD_TRUE. Include **tti_time** in the PDU.

tti_time This structure of type **EVENT_TIME** contains the time of the last transition to IDLE.

iack_time_pres **SD_FALSE**. Do NOT include **iack_time** in the PDU.

SD_TRUE. Include **iack_time** in the PDU.

iack_time This structure of type **EVENT_TIME** exists only for Event Enrollments that reference a monitored Event Condition. Its value is only significant when the value of the Alarm Acknowledgment Rule (**alarm_ack_rule**) is NOT equal to NONE.

It records the time (date and time or Time Sequence Identifier). This is the time at which an acknowledgment for the most recently detected transition of the Event Condition to the IDLE state was received from the enrolled client. If this acknowledgment was not received, this value equals UNDEFINED.

NOTES:

1. Include this structure only if there has been a corresponding transition and:
2. the Event Condition’s alarm summary reports attribute is TRUE, or
3. if the Event Condition is referenced by an Event Enrollment whose alarm acknowledgment rule is not NONE.

Paired Primitive Interface Functions

mp_getaes

Usage: This primitive request function sends a GetAlarmEnrollmentSummary request PDU to a remote node. It uses the data found in a structure of type **GETAES_REQ_INFO** pointed to by **info**. This service is used to obtain summary information about the status of various Event Enrollments at a remote VMD.

Function Prototype:

```
MMSREQ_PEND *mp_getaes (ST_INT chan,
                        GETAES_REQ_INFO *info);
```

Parameters:

chan	This is the channel number over which the PDU is to be sent.
info	This pointer to the operation-specific data structure of type **GETAES_REQ_INFO** contains data specific to the request PDU to be sent.

Return Value:

MMSREQ_PEND *	This pointer to the request control data structure of type **MMSREQ_PEND** is used to send the PDU. In case of an error, the pointer is set to null and **mms_op_err** is written with the error code.

Corresponding User Confirmation Function: **u_mp_getaes_conf**

Operation-Specific Data Structure Used: **GETAES_REQ_INFO**

See page 3-187 for a detailed description of this structure.

u_getaes_ind

Usage: This user indication function is called when a GetAlarmEnrollmentSummary indication is received by a server node. The contents and operation of this function are user-defined, and depend on the application requirements. To conform to the MMS protocol requirements, the application must do one of the following some time after the indication is received:

1) send a positive response using the primitive response function (**`mp_getaes_resp`**) after the specified Alarm Enrollment Summary information has been successfully obtained, or

2) call the appropriate primitive negative (error) response function (**`mp_err_resp`**). See **Module 11 — MMS-EASE Error Handling** on page 3-343 for an explanation of this function.

See **Volume 1 — Module 1 — User Indication Function Class** for additional information regarding recommended handling of indications.

Function Prototype: `ST_VOID u_getaes_ind (MMSREQ_IND *ind);`

Parameters:

`ind` This pointer to the **`MMSREQ_IND`** structure is described in **Volume 1 — Module 2 — Request and Indication Control Data Structures**. This pointer must be used to send the response to this indication. The primary item of interest in this structure is **`ind->req_info_ptr`**. This is a pointer to the operation-specific data structure for a GetAlarmEnrollmentSummary indication (**`GETAES_REQ_INFO`**). This pointer will always be valid when **`u_getaes_ind`** is called.

Return Value: `ST_VOID` (ignored)

Operation-Specific Data Structure Used: **`GETAES_REQ_INFO`**

See page 2-187 for a detailed description of this structure.

mp_getaes_resp

Usage: This primitive response function sends a GetAlarmEnrollmentSummary positive response PDU. It uses the data from a structure of type **GETAES_RESP_INFO**, pointed to by **info**. This function should be called as a response to the **u_getaes_ind** function being called (a GetAlarmEnrollmentSummary indication is received), and after the specified Alarm Enrollment Summary information has been successfully obtained.

Function Prototype:

```
ST_RET mp_getaes_resp (MMSREQ_IND *ind,
                       GETAES_RESP_INFO *info);
```

Parameters:

`ind` This pointer to the indication control structure of type **MMSREQ_IND** is received in the **u_getaes_ind** function.

`info` This pointer to an Operation-Specific data structure of type **GETAES_RESP_INFO** contains information specific to the response PDU to be sent.

Return Value: `ST_RET`

SD_SUCCESS. No Error.

`<> 0` Error Code.

Corresponding User Indication Function: **u_getaes_ind**

Operation-Specific Data Structure Used: **GETAES_RESP_INFO**

See page 3-188 for a detailed description of this structure.

u_mp_getaes_conf

Usage: This primitive user confirmation function is called when a confirm to a GetAlarmEnrollmentSummary request (**mp_getaes**) is received. **resp_err** contains a value indicating whether an error occurred. **resp_info_ptr** may contain data that the remote node sent back in the confirmation or it may contain error information if an error occurred. If no error has occurred (**req->resp_err = CNF_RESP_OK**), GetAlarmSummary information is available to the application using the **req_ptr->resp_info_ptr** pointer. See **Volume 1 — Module 1 — Primitive User Confirmation Function Class** for specific information on the recommended handling of confirms.

Note that the function called for this event may be changed if desired; however, **u_mp_getaes_conf** is the standard default function.

Function Prototype:

```
ST_VOID u_mp_getaes_conf (MMSREQ_PEND *req);
```

Parameters:

`req` This pointer to the **MMSREQ_PEND** structure is returned from the original **mp_getaes** request. See **Volume 1 — Module 2 — Request and Indication Control Data Structures** for information on the structure.

For a positive response (**req->resp_err = CNF_RESP_OK**), the request control structure member **resp_info_ptr** is a pointer to the operation-specific data structure (**GETAES_RESP_INFO**) for a GetAlarmEnrollmentSummary function.

For a negative response, or other errors, see **Module 11 — MMS-EASE Error Handling** starting on page 3-323 for more information.

Return Value: `ST_VOID` (ignored)

Operation-Specific Data Structure Used: **GETAES_RESP_INFO**

See page 3-188 for a detailed description of this structure.

32. Attach to Event Condition Modifier

The **attach_to_evcon** modifier is provided so that processing can be delayed until a specified event condition object undergoes any one of a specified set of state transitions. This may be used to modify the execution of a service or to modify the execution of an event action. See page 3-1 for an explanation of general modifier handling for all confirmed requests.

```
struct attach_to_evcon
  {
  OBJECT_NAME evenroll_name;
  OBJECT_NAME evcon_name;
  ST_UCHAR    causing_transitions;
  ST_BOOLEAN  acc_delay_pres;
  ST_UINT32   acc_delay;
  SD_END_STRUCT
  };
typedef struct attach_to_evcon ATTACH_TO_EVCON;
```

Fields:

`evenroll_name` This structure of type **OBJECT_NAME** contains the name of the Event Enrollment.

`evcon_name` This structure of type **OBJECT_NAME** contains the name of the Event Condition.

`causing_transitions` This bitstring indicates the transitions that trigger the execution of the modifier service:

- **0** IDLE-TO-DISABLED
- **1** ACTIVE-TO-DISABLED
- **2** DISABLE-TO-IDLE
- **3** ACTIVE to IDLE
- **4** DISABLED-TO-IDLE
- **5** IDLE-TO-ACTIVE
- **6** ANY-TO-DELETED

`acc_delay_pres` **SD_TRUE**. Include **acc_delay** in the PDU.

SD_FALSE. Do Not include **acc_delay** in the PDU.

`acc_delay` This indicates the number of milliseconds of acceptable delay the remote node should wait for Event Condition object's state to transition before sending an error response. The default is wait indefinitely.

NOTE: See **Volume 1 — Module 2 — MMS Object Name Structure** for a detailed description of this structure.

33. Journal Management Introduction

A MMS journal represents a log file that contains a collection of records called *journal entries* that are organized by time stamps. Journal are used to store time based records of tagged variable data, user generated comments, or combinations of events and tagged variable data. Journal entries contain a time stamp that indicates when the data in the entry was produced (not when the journal entry was made). This allows MMS journals to be used for applications where a sample of manufactured product is taken at one time, analyzed in a laboratory off-line and then at a later time, placed into the journal. In this case, the journal entry time stamp would indicate when the sample was taken.

MMS clients read journal entries by specifying the name of the journal (which can be VMD-specific or AA-specific only) and either

1. the date/time range of entries that the client wishes to read, or
2. by referring to the entry ID of a particular entry.

This entry ID is a unique binary identifier assigned by the VMD to the journal entry when it is placed into the journal. Each entry in a journal can be one of the following types:

Annotation This type of entry contains textual comment. This is typically used to enter a comment regarding some event or condition that had occurred in the system.

Data This type of entry contains a list of variable tags and the data associated with those tags at the time indicated by the time stamp. Each variable tag is a 32-character name that does not necessarily refer to a MMS variable (although it might).

Event-Data This type of entry contains both variable tag data and event data. Each entry of this type would include the same list of variable tags and associated data as described above along with a single event condition name and the state of that event condition at the time indicated by the time stamp.

The six services available for MMS journals are as follows:

ReadJournal This service is used by a client to read one or more entries from a journal. See page 3-203 for more information.

WriteJournal This service is used by a client to create new journal entries in a journal. A journal entry can also be created by local autonomous action by the VMD without a client using the WriteJournal service. See page 3-211 for more information.

InitializeJournal This service is used by a client to delete all or some of the journal entries in a journal. For instance, a client can use InitializeJournal to delete old journal entries that are no longer of any interest. See page 3-217 for more information.

ReportJournalStatus This service is used to obtain the number of journal entries in a journal. See page 3-223 for more information.

CreateJournal This service is used by a client to create a journal object. The CreateJournal service only creates the journal; it does not create any journal entries (see ReadJournal). See page 3-229 for more information.

DeleteJournal This service is used by a client to delete a journal object if the journal is deletable. See page 3-235 for more information.

Common Journal Data Structures

Below find the common data structures used by Journal Management services.

Variable Information Structure

The following structure holds variable information for a Journal Entry

```
struct var_info
  {
  ST_CHAR      *var_tag;
  VAR_ACC_DATA value_spec;
  };
typedef struct var_info   VAR_INFO;
```

Fields:

`var_tag` This is a pointer to a name to be given to the data that follows in **`value_spec`**. This can be comprised of no more than 32 characters of a Visible String only.

`value_spec` This structure of type **`VAR_ACC_DATA`** contains the data corresponding to the name given in **`var_tag`**. See **Volume 2 — Module 5 — Variable Access Result Structures** for a detailed description of this structure.

Journal Content Structure

The following structure specifies the contents of a journal entry:

```
struct entry_content
  {
  MMS_BTOD    occur_time;
  ST_BOOLEAN  addl_detail_pres;
  ST_INT      addl_detail_len;
  ST_UCHAR    *addl_detail;
  ST_INT16    entry_form_tag;
  union
    {
    struct
      {
      ST_BOOLEAN  event_pres;
      OBJECT_NAME evcon_name;
      ST_INT16    cur_state;
      ST_BOOLEAN  list_of_var_pres;
      ST_INT      num_of_var;
      } data;
    ST_CHAR    *annotation;
    }ef;
/*VAR_INFO    list_of_var [num_of_var];                          */
  SD_END_STRUCT
  };
typedef struct entry_content ENTRY_CONTENT;
```

Fields:

`occur_time` This structure of type **`MMS_BTOD`** contains the occurrence time of this Journal entry. See page 3-5 for a detailed description of this structure.

`addl_detail_pres` **`SD_FALSE`**. Do NOT include **`addl_detail_len`** and **`addl_detail`** in the PDU.

`SD_TRUE`. Include **`addl_detail_len`** and **`addl_detail`** in the PDU.

`addl_detail_len` This is the length, in bytes, of the data pointed to by **`addl_detail`**. Use only if **`addl_detail_pres != 0`**.

`addl_detail` This contains the Additional Detail. The content and form of this data must be valid ASN.1.

`entry_form_tag` This is a tag indicating the form of the Journal Entry:

`2` **DATA** (default)
`3` **ANNOTATION**

`event_pres` **`SD_FALSE`**. Do NOT include **`evcon_name`** and **`cur_state`** in the PDU.

`SD_TRUE`. Include **`evcon_name`** and **`cur_state`** in the PDU.

`evcon_name` This structure of type **`OBJECT_NAME`** contains the name of the Event Condition corresponding to this Journal Entry. Use only if **`(entry_form_tag = 2)&&(event_pres = SD_FALSE)`**. See **Volume 1 — Module 2 — MMS Object Name Structure** for an explanation of this structure.

`cur_state` This is the current state of the Event Condition. Use only if **`(entry_form = 2)&& (event_pres = SD_TRUE)`**.

`0` **DISABLED**
`1` **IDLE**
`2` **ACTIVE**

`list_of_var_pres` **`SD_FALSE`**. Do NOT include **`num_of_var`** and **`list_of_var`** in the PDU.

`SD_TRUE`. Include **`num_of_var`** and **`list_of_var`** in the PDU.

`num_of_var` This is the number of entries in **`list_of_var`**. Use only if **`(entry_form = 2)&& (list_of_var_pres = SD_TRUE)`**.

`annotation` This is a pointer to the annotation form of this Journal Entry. It should be a Visible String containing comments or description of the Journal Entry. Use only if **`(entry_form = 3)`**.

`list_of_var` This array of structures of type **`VAR_INFO`** contains a list of variables to be included in the Journal Entry. Use only if **`(entry_form = 2)&&(list_of_var_pres = SD_TRUE)`**. See the previous page for an explanation of this structure.

NOTES:

1. The union **`ef`** in the **`ENTRY_CONTENT`** structure holds either a pointer to the annotation (if the form of the journal entry is annotation) or it holds a structure of type data (if the form of the journal entry is data). The value of **`entry_content.entry_form_tag`** determines whether the form of the journal entry is data or annotation.

2. When allocating a structure of type **`ENTRY_CONTENT`**, enough memory must be allocated to hold the list of variables contained in **`list_of_var`**. The following C language statement can be used:

```
info = (ENTRY_CONTENT *) chk_malloc(sizeof (ENTRY_CONTENT) + (num_of_var *
            sizeof(VAR_INFO)));
```

Journal Entry Structure

The following structure specifies a journal entry. Each entry of a journal, which can consist of multiple journal entries, uses a structure of this type:

```
struct journal_entry
  {
  ST_INT        entry_id_len;
  ST_UCHAR      *entry_id;
  ST_INT        orig_ae_len;
  ST_UCHAR       *orig_ae;
  ENTRY_CONTENT ent_content;
  };
typedef struct journal_entry JOURNAL_ENTRY;
```

Fields:

`entry_id_len` This is the length, in bytes, of the entry identifier pointed to by **`entry_id`**.

`entry_id`	This is a pointer to the entry identifier assigned to this journal entry by the MMS server responsible for managing this journal.
`orig_ae_len`	This is the length, in bytes, of the data pointed to by **`orig_ae`**.
`orig_ae`	This pointer to an ASN.1 encoded Object Identifier corresponds to the application entity that originally created this journal entry.
`ent_content`	This structure of type **`ENTRY_CONTENT`** includes the contents of this journal entry.

34. ReadJournal Service

This service is used by the client to request that a server retrieve information out of a specified Journal object, and return this information to the client. If the entire Journal object contents cannot be returned, the client may specify various filters that can be used. The contents of the Journal object is not affected by this service.

Primitive Level ReadJournal Operations

The following section contains information on how to use the paired primitive interface for the ReadJournal service. It covers available data structures used by the PPI, and the four primitive level functions that together make up the ReadJournal service. See **Volume 1 — Module 2 — General Sequence of Events — Confirmed Services** for a detailed explanation of the sequence of events that occur within the MMS-EASE environment during confirmed services.

- The ReadJournal service consists of the paired primitive functions of **`mp_jread`**, **`u_jread_ind`**, **`mp_jread_resp`**, and **`u_mp_jread_conf`**.

Data Structures

Request/Indication

The operation-specific structure described below is used by the client in issuing a ReadJournal request (**`mp_jread`**). It is received by the server when a ReadJournal indication (**`u_jread_ind`**) is received.

```
struct jread_req_info
  {
  OBJECT_NAME jou_name;
  ST_BOOLEAN  range_start_pres;
  ST_INT16    start_tag;
  MMS_BTOD    start_time;
  ST_INT      start_entry_len;
  ST_UCHAR    *start_entry;
  ST_BOOLEAN  range_stop_pres;
  ST_INT16    stop_tag;
  MMS_BTOD    end_time;
  ST_INT32    num_of_entries;
  ST_BOOLEAN  list_of_var_pres;
  ST_INT      num_of_var;
  ST_BOOLEAN  sa_entry_pres;
  MMS_BTOD    time_spec;
  ST_INT      entry_spec_len;
  ST_UCHAR    *entry_spec;
/*ST_CHAR     *list_of_var [num_of_var];                              */
  SD_END_STRUCT
  };
typedef struct jread_req_info JREAD_REQ_INFO;
```

Fields:

`jou_name` This structure of type **`OBJECT_NAME`** contains the name of the journal to read. See **Volume 1 — Module 2 — MMS Object Name Structure** for an explanation of this structure.

`range_start_pres` **`SD_FALSE`**. Do not include **`start_tag`**, **`start_time`**, **`start_entry_len`** or **`start_entry`** in the PDU.

`SD_TRUE`. Include **`start_tag`**, **`start_time`**, **`start_entry_len`** and **`start_entry`** in the PDU.

start_tag	**0**	Read Journal Entries that are younger than **start_time**.
	1	Read Journal Entries after the first entry that matches **start_entry**.
start_time	This structure of type **MMS_BTOD** contains the time to start reading the Journal Entries. See page 3-5 for a detailed description of this structure.	
start_entry_len	This is the length, in bytes, of the data pointed to by **start_entry**.	
start_entry	This is a pointer to the entry identifier after which to start the read. This data contains an entry identifier, an octet string of no more than 8 octets (bytes), specific to the VMD. It contains the journal and is used to specify unique multiple journal entries having the same time entry.	
range_stop_pres	**SD_FALSE**. Do not include **end_time** or **num_of_entries** in the PDU.	
	SD_TRUE. Include **end_time** or **num_of_entries** in the PDU as specified by **stop_tag**.	
stop_tag	**0**	Use **end_time**.
	1	Use **num_of_entries**.
end_time	This structure of type **MMS_BTOD** contains the end time. Do not read any entries younger than the specified time. See page 3-5 for a detailed description of this structure.	
num_of_entries	This contains the number of entries to read. Read only the specified number of entries regardless of the end time.	
list_of_var_pres	**SD_FALSE**. Do NOT include the **list_of_var** field in the PDU.	
	SD_TRUE. Include the **list_of_var** field in the PDU.	
num_of_var	This indicates the number of variable tags in the **list_of_var** array.	
sa_entry_pres	**SD_FALSE**. Do NOT include **time_spec** or **entry_spec** in the PDU. This tells the remote node to begin the ReadJournal response with the first entry matching the start and stop specifications described above.	
	SD_TRUE. Include **time_spec** and **entry_spec** in the PDU. These specify where the remote node should begin its ReadJournal response for later requests when the entire list requested could not be returned in a single request. Use only if this is a subsequent ReadJournal request after a response has indicated **more_follows**.	
time_spec	This structure of type **MMS_BTOD** specifies the entry time to start after for chained requests. This is used in subsequent ReadJournal requests if the entire list of journal entries could not be returned in the first request. Use only if this is a subsequent ReadJournal request after a response has indicated **more_follows**. See page 3-208 for more information on **mp_jread_resp**.	
entry_spec_len	This is the length, in bytes, of the data pointed to by **entry_spec**.	
entry_spec	This specifies the entry identifier after which to start the read. This data contains an entry identifier. This is an octet string of no more than eight octets (bytes) specific to the VMD that contains the journal. It is used to specify unique multiple journal entries having the same entry time. Use only if this is a later ReadJournal request after a response has indicated **more_follows**.	
list_of_var	This specifies the variable tags (names) for which the journal entries are to be read. Only those journal entries containing these specified variables will be returned.	

NOTE: When allocating a structure of type **JREAD_REQ_INFO**, enough memory must be allocated to hold the list of variables member (**list_of_var**) of this structure and the variable tags themselves pointed to by **list_of_var**. The following C language statement can be used to allocate the memory needed by this structure. However, this will not allocate the memory to hold the actual variable tags themselves, only the pointers to the variable tags contained in **list_of_var**.

```
info = (JREAD_REQ_INFO *) chk_malloc(sizeof(JREAD_REQ_INFO) + (num_of_var *
            sizeof(ST_CHAR *)));
```

Response/Confirm

The operation-specific data structure described below is used by the server in issuing a ReadJournal response (**mp_jread_resp**). It is received by the client when a ReadJournal confirm (**u_mp_jread_conf**) is received.

```
struct jread_resp_info
  {
  ST_INT        num_of_jou_entry;
  ST_BOOLEAN    more_follows;
/*JOURNAL_ENTRY list_of_jou_entry [num_of_jou_entry];                    */
  SD_END_STRUCT
  };
typedef struct jread_resp_info JREAD_RESP_INFO;
```

Fields:

num_of_jou_entry	This indicates the number of Journal Entries in this Journal.
more_follows	**SD_TRUE**. There are more Journal Entries available. **SD_FALSE**. This is the end of the Journal Entries.
list_of_jou_entry	This array of structures of type **JOURNAL_ENTRY** contains information regarding each Journal Entry in the response or confirm. See page 3-202 for more information on this structure.

NOTE: When allocating a data structure of type **JREAD_RESP_INFO**, enough memory must be allocated to hold the information for the array of structures containing the Journal Entry list in the **list_of_jou_entry[]** member of this structure. The following C statement can be used:

```
info = (JREAD_RESP_INFO *) chk_malloc(sizeof(JREAD_RESP_INFO)+ num_of_jou_entry *
            sizeof(JOURNAL_ENTRY);
```

Paired Primitive Interface

mp_jread

Usage: This primitive request function sends a ReadJournal request PDU to a remote node. It uses the data found in a structure of type **JREAD_REQ_INFO** pointed to by **info**. This service is used to read the entries of a Journal at the remote node.

Function Prototype:

```
MMSREQ_PEND *mp_jread (ST_INT chan,
                       JREAD_REQ_INFO *info);
```

Parameters:

chan	This is the channel number over which the PDU is to be sent.
info	This pointer to the operation-specific data structure of type **JREAD_REQ_INFO** contains data specific to the request PDU to be sent.

Return Value:

MMSREQ_PEND *	This pointer to the request control data structure of type **MMSREQ_PEND** is used to send the PDU. In case of an error, the pointer is set to null and **mms_op_err** is written with the error code.

Corresponding User Confirmation Function: **u_mp_jread_conf**

Operation-Specific Data Structure Used: **JREAD_REQ_INFO**

See page 3-203 for a detailed description of this structure.

u_jread_ind

Usage: This user indication function is called when a ReadJournal indication is received by a server node. The contents and operation of this function are user-defined, and depend on the application requirements. To conform to the MMS protocol requirements, the application must do one of the following some time after the indication is received:

1) send a positive response using the primitive response function (**`mp_jread_resp`**) after the specified Journal Entries have been successfully read, or

2) call the appropriate primitive negative (error) response function (**`mp_err_resp`**). See **Module 11 — MMS-EASE Error Handling** on page 3-343 for an explanation of this function.

See **Volume 1 — Module 1 — User Indication Function Class** for additional information regarding recommended handling of indications.

Function Prototype: `ST_VOID u_jread_ind (MMSREQ_IND *ind);`

Parameters:

`ind` This pointer to the **`MMSREQ_IND`** structure is described in **Volume 1 — Module 2 — Request and Indication Control Data Structures**. This pointer must be used to send the response to this indication. The primary item of interest in this structure is **`ind->req_info_ptr`**. This is a pointer to the operation-specific data structure for a ReadJournal indication (**`JREAD_REQ_INFO`**). This pointer will always be valid when **`u_jread_ind`** is called.

Return Value: `ST_VOID` (ignored)

Operation-Specific Data Structure Used: **`JREAD_REQ_INFO`**

See page 3-203 for a detailed description of this structure.

mp_jread_resp

Usage: This primitive response function sends a ReadJournal positive response PDU using the data from a structure of type **JREAD_RESP_INFO**, pointed to by **info**. This function should be called as a response to the **u_jread_ind** function being called (a ReadJournal indication is received), and after the specified Journal Entries have been successfully read.

Function Prototype:

```
ST_RET mp_jread_resp (MMSREQ_IND *ind,
                      JREAD_RESP_INFO *info);
```

Parameters:

`ind` This pointer to the indication control structure of type **MMSREQ_IND** is received in the **u_jread_ind** function.

`info` This pointer to an Operation Specific data structure of type **JREAD_RESP_INFO** contains information specific to the response PDU to be sent.

Return Value: `ST_RET` **SD_SUCCESS**. No Error.

`<> 0` Error Code.

Corresponding User Indication Function: **u_jread_ind**

Operation-Specific Data Structure Used: **JREAD_RESP_INFO**

See page 3-205 for a detailed description of this structure.

u_mp_jread_conf

Usage: This primitive user confirmation function is called when a confirm to a ReadJournal request (**mp_jread**) is received. **resp_err** contains a value indicating whether an error occurred. **resp_info_ptr** may contain data that the remote node sent back in the confirmation or it may contain error information if an error occurred. If no error has occurred (**req->resp_err = CNF_RESP_OK**), ReadJournal information is available to the application using the **req->resp_info_ptr** pointer. See **Volume 1 — Module 1 — Primitive User Confirmation Function Class** for specific information on the recommended handling of confirms.

Note that the function called for this event may be changed if desired; however, **u_mp_jread_conf** is the standard default function.

Function Prototype:

```
ST_VOID u_mp_jread_conf (MMSREQ_PEND *req);
```

Parameters:

`req` This pointer to the **MMSREQ_PEND** structure is returned from the original **mp_jread** request. See **Volume 1 — Module 2 — Request and Indication Control Data Structures** for information on the structure.

For a positive response (**req->resp_err = CNF_RESP_OK**), the request control structure member **resp_info_ptr** is a pointer to the operation specific data structure (**JREAD_RESP_INFO**) for the ReadJournal function.

For a negative response, or other errors, see **Module 11 — MMS-EASE Error Handling** starting on page 3-339 for more information.

Return Value: `ST_VOID` (ignored)

Operation-Specific Data Structure Used: **JREAD_RESP_INFO**

See page 3-205 for a detailed description of this structure.

35. WriteJournal Service

This service is used by the client to place one or more journal entries in a Journal object.

Primitive Level WriteJournal Operations

The following section contains information on how to use the paired primitive interface for the WriteJournal service. It covers available data structures used by the PPI, and the four primitive level functions that together make up the WriteJournal service. See **Volume 1 — Module 2 — General Sequence of Events — Confirmed Services** for a detailed explanation of the sequence of events that occur within the MMS-EASE environment during confirmed services.

- The WriteJournal service consists of the paired primitive functions of **`mp_jwrite`**, **`u_jwrite_ind`**, **`mp_jwrite_resp`**, and **`u_mp_jwrite_conf`**.

Data Structures

Request/Indication

The operation-specific structure described below is used by the client in issuing a WriteJournal request (**`mp_jwrite`**). It is received by the server when a WriteJournal indication (**`u_jwrite_ind`**) is received.

```
struct jwrite_req_info
  {
  OBJECT_NAME   jou_name;
  ST_INT        num_of_jou_entry;
/*ENTRY_CONTENT list_of_jou_entry [num_of_jou_entry];                    */
  SD_END_STRUCT
  };
typedef struct jwrite_req_info JWRITE_REQ_INFO;
```

Fields:

`jou_name` This structure of type **`OBJECT_NAME`** contains the name of the Journal to write. See **Volume 1 — Module 2 — MMS Object Name Structure** for a detailed description of this structure.

`num_of_jou_entry` This indicates the number of Journal Entries to write.

`list_of_jou_entry` This array of structures of type **`ENTRY_CONTENT`** contains the contents of the entries to write. See page 3-200 for a detailed description of this structure.

NOTE: When allocating a structure of type **`JWRITE_REQ_INFO`**, enough memory must be allocated to hold the journal entry contents, **`list_of_jou_entry`**. The following C language statement can be used:

```
info = (JWRITE_REQ_INFO *) chk_malloc(sizeof(JWRITE_REQ_INFO) + (num_of_jou_entry
            * sizeof(ENTRY_CONTENT)) + (num_of_var * sizeof(VAR_INFO)));
```

Paired Primitive Interface Functions

mp_jwrite

Usage: This primitive request function sends a WriteJournal request PDU to a remote node. It uses the data found in a structure of type **JWRITE_REQ_INFO**, pointed to by **info**. This service is used to write the contents of Journal Entry at the remote node.

Function Prototype:

```
MMSREQ_PEND *mp_jwrite (ST_INT chan,
                        JWRITE_REQ_INFO *info);
```

Parameters:

chan	This is the channel number over which the PDU is to be sent.
info	This pointer to the operation-specific data structure of type **JWRITE_REQ_INFO** contains data specific to the request PDU to be sent.

Return Value:

MMSREQ_PEND *	This pointer to the Queue data structure of type **MMSREQ_PEND** is used to send the PDU. In case of an error, the pointer is set to null and **mms_op_err** is written with the error code.

Corresponding User Confirmation Function: **u_mp_jwrite_conf**

Operation-Specific Data Structure Used: **JWRITE_REQ_INFO**

See page 3-211 for a detailed description of this structure.

u_jwrite_ind

Usage: This user indication function is called when a WriteJournal indication is received by a server node. The contents and operation of this function are user-defined, and depend on the application. To conform to the MMS protocol, the application must do one of the following some time after the indication is received:

1) send a positive response using the primitive response function (**`mp_jwrite_resp`**) after the specified Journal information has been successfully written to the Journal, or

2) call the appropriate primitive negative (error) response function (**`mp_err_resp`**). See **Module 11 — MMS-EASE Error Handling** on page 3-343 for an explanation of this function.

See **Volume 1 — Module 1 — User Indication Function Class** for additional information regarding the recommended handling of indications.

Function Prototype:

```
ST_VOID u_jwrite_ind (MMSREQ_IND *ind);
```

Parameters:

`ind` This pointer to the **`MMSREQ_IND`** structure is described in **Volume 1 — Module 2 — Request and Indication Control Data Structures**. This pointer must be used to send the response to this indication. The primary item of interest in this structure is **`ind->req_info_ptr`**. This is a pointer to the operation-specific data structure for a WriteJournal indication (**`JWRITE_REQ_INFO`**). This pointer will always be valid when **`u_jwrite_ind`** is called.

Return Value: `ST_VOID` (ignored)

Operation Specific Data Structure Used: **`JWRITE_REQ_INFO`**

See page 3-211 for a detailed description of this structure.

mp_jwrite_resp

Usage: This primitive response function sends a WriteJournal positive response PDU. This function should be called after the **`u_jwrite_ind`** function is called (a WriteJournal indication is received), and after the specified Journal information has been successfully written to the Journal.

Function Prototype: `ST_RET mp_jwrite_resp (MMSREQ_IND *ind);`

Parameters:

`ind` This is pointer to the indication control structure of type **`MMSREQ_IND`** is received in the **`u_jwrite_ind`** function.

Return Value: `ST_RET` **`SD_SUCCESS`**. No Error.

`<> 0` Error Code.

Corresponding User Indication Function: **`u_jwrite_ind`**

Operation-Specific Data Structure Used: NONE

u_mp_jwrite_conf

Usage: This primitive user confirmation function is called when a confirm to a WriteJournal request (**mp_jwrite**) is received. **resp_err** contains a value indicating whether an error occurred. **resp_info_ptr** may contain data that the remote node sent back in the confirmation or it may contain error information if an error occurred. See **Volume 1 — Module 1 — Primitive User Confirmation Function Class** for specific information on the recommended handling of confirms.

Note that the function called for this event may be changed if desired; however, **u_mp_jwrite_conf** is the standard default function.

Function Prototype:

```
ST_VOID u_mp_jwrite_conf (MMSREQ_PEND *req);
```

Parameters:

`req` This pointer to the **MMSREQ_PEND** structure is returned from the original **mp_jwrite** request. See **Volume 1 — Module 2 — Request and Indication Control Data Structures** for information on the structure.

For a positive response (**req->resp_err = CNF_RESP_OK**), the request control structure member **resp_info_pres** indicates that there is no confirm data for this function.

For a negative response, or other errors, see **Module 11 — MMS-EASE Error Handling** starting on page 3-339 for more information.

Return Value: `ST_VOID` (ignored)

Operation Specific Data Structure Used: NONE

36. InitializeJournal Service

This service is used by the client to request that a server initialize all or part of an existing Journal object by removing all or some of the journal entries.

Primitive Level InitializeJournal Operations

The following section contains information on how to use the paired primitive interface for the InitializeJournal service. It covers available data structures used by the PPI, and the four primitive level functions that together make up the InitializeJournal service. See **Volume 1 — Module 2 — General Sequence of Events — Confirmed Services** for a detailed explanation of the sequence of events that occur within the MMS-EASE environment during confirmed services.

- The InitializeJournal service consists of the paired primitive functions of **`mp_jinit`**, **`u_jinit_ind`**, **`mp_jinit_resp`**, and **`u_mp_jinit_conf`**.

Data Structures

Request/Indication

The operation-specific structure described below is used by the client in issuing an InitializeJournal request (**`mp_jinit`**). It is received by the server when an InitializeJournal indication (**`u_jinit_ind`**) is received.

```
struct jinit_req_info
  {
  OBJECT_NAME jou_name;
  ST_BOOLEAN  limit_spec_pres;
  MMS_BTOD limit_time;
  ST_BOOLEAN  limit_entry_pres;
  ST_INT      limit_entry_len;
  ST_UCHAR    *limit_entry;
  SD_END_STRUCT
  };
typedef struct jinit_req_info JINIT_REQ_INFO;
```

Fields:

`jou_name` This structure of type **`OBJECT_NAME`** contains the name of the journal to be initialized. See **Volume 1 — Module 2 — MMS Object Name Structure** for a detailed description of this structure.

`limit_spec_pres` **`SD_FALSE`**. Do NOT include **`limit_time`** or **`limit_entry`** in PDU. All Journal Entries will be cleared.

`SD_TRUE`. Include at least **`limit_time`** in the PDU. Examine **`limit_entry_pres`** to determine whether to include **`limit_entry`** in the PDU.

`limit_time` This structure of type **`MMS_BTOD`** specifies the time limit used to determine which Journal Entries are to be initialized. Only those Journal Entries that are older than the specified time will be initialized.

limit_entry_pres	**SD_FALSE**. Do NOT include **limit_entry** in PDU. Journal Entries cleared will be based on **limit_time** only. **SD_TRUE**. Include **limit_entry** in the PDU.
limit_entry_len	This is the length, in bytes, of the data pointed to by **limit_entry**.
limit_entry	This pointer to the Limiting Entry Specifier contains an entry identifier that is an octet string of no more than eight octets (bytes). It is used to resolve multiple entries that have the same occurrence time. The form of the entry specifier is dependent on the particular VMD. This contains the Journal and contains an octet string used to specify unique multiple journal entries that have the same time entry.

Response/Confirm

The operation-specific data structure described below is used by the server in issuing an InitializeJournal response (**mp_jinit_resp**). It is received by the client when an InitializeJournal confirm (**u_mp_jinit_conf**) is received.

```
struct jinit_resp_info
  {
  ST_UINT32  del_entries;
  };
typedef struct jinit_resp_info JINIT_RESP_INFO;
```

Fields:

del_entries	This indicates the number of journal entries that were deleted as a successful result of the InitializeJournal service request.

Paired Primitive Interface Functions

mp_jinit

Usage: This primitive request function sends an InitializeJournal request PDU to a remote node. It uses the data found in a structure of type **JINIT_REQ_INFO**, pointed to by **info**. This service is used to tell the remote node to initialize all or part of a Journal.

Function Prototype:

```
MMSREQ_PEND *mp_jinit (ST_INT chan,
                       JINIT_REQ_INFO *info);
```

Parameters:

chan	This is the channel number over which the PDU is to be sent.
info	This pointer to the Operation-Specific data structure of type **JINIT_REQ_INFO** contains data specific to the request PDU to be sent.

Return Value:

MMSREQ_PEND *	This pointer to the request control data structure of type **MMSREQ_PEND** is used to send the PDU. In case of an error, the pointer is set to null and **mms_op_err** is written with the error code.

Corresponding User Confirmation Function: **u_mp_jinit_conf**

Operation-Specific Data Structure Used: **JINIT_REQ_INFO**

See page 3-217 for a detailed description of this structure.

u_jinit_ind

Usage: This user indication function is called when an InitializeJournal indication is received by a server node. The contents and operation of this function are user-defined, and depend on the application requirements. To conform to MMS protocol, the application must do one of the following some time after the indication is received:

1) send a positive response using the primitive response function (**mp_jinit_resp**) after the specified Journal Entries have been initialized (cleared), or

2) call the appropriate primitive negative (error) response function (**mp_err_resp**). See **Module 11 — MMS-EASE Error Handling** on page 3-343 for an explanation of this function.

See **Volume 1 — Module 1 — User Indication Function Class** for additional information regarding the recommended handling of indications.

Function Prototype:

```
ST_VOID u_jinit_ind  (MMSREQ_IND *ind);
```

Parameters:

`ind` This pointer to the **MMSREQ_IND** structure is described in **Volume 1 — Module 2 — Request and Indication Control Data Structures**. This pointer must be used to send the response to this indication. The primary item of interest in this structure is **ind->req_info_ptr**. This is a pointer to the operation-specific data structure for an InitializeJournal indication (**JINIT_REQ_INFO**). This pointer will always be valid when **u_jinit_ind** is called.

Return Value: `ST_VOID` (ignored)

Operation-Specific Data Structure Used: **JINIT_REQ_INFO**

See page 3-217 for a detailed description of this structure.

mp_jinit_resp

Usage: This primitive response function sends an InitializeJournal positive response PDU using the data from a structure of type **JINIT_RESP_INFO**, pointed to by **info**. This function should be called as a response to the **u_jinit_ind** function being called (an InitializeJournal indication is received), and after the specified Journal Entries have been initialized (cleared).

Function Prototype:

```
ST_RET mp_jinit_resp (MMSREQ_IND *ind,
                      JINIT_RESP_INFO *info);
```

Parameters:

`ind` This pointer to the indication control structure of type **MMSREQ_IND** is received in the **u_jinit_ind** function.

`info` This pointer to an Operation Specific data structure of type **JINIT_RESP_INFO** contains information specific to the response PDU to be sent.

Return Value: `ST_RET` **SD_SUCCESS**. No Error.

`<> 0` Error Code.

Corresponding User Indication Function: **u_jinit_ind**

Operation-Specific Data Structure Used: **JINIT_RESP_INFO**

See page 3-218 for a detailed description of this structure.

u_mp_jinit_conf

Usage: This primitive user confirmation function, is called when a confirm to an InitializeJournal request (**mp_jinit**) is received. **resp_err** contains a value indicating whether an error occurred. **resp_info_ptr** may contain data that the remote node sent back in the confirmation or it may contain error information if an error occurred. If no error has occurred (**req->resp_err = CNF_RESP_OK**), InitializeJournal information is available to the application using the **req->resp_info_ptr** pointer. See **Volume 1 — Module 1 — Primitive User Confirmation Function Class** - for specific information on the recommended handling of confirms.

Note that the function called for this event may be changed if desired; however, **u_mp_jinit_conf** is the standard default function.

Function Prototype: `ST_VOID u_mp_jinit_conf (MMSREQ_PEND *req);`

Parameters:

`req` This pointer to the **MMSREQ_PEND** structure is returned from the original **mp_jinit** request. See **Volume 1 — Module 2 — Request and Indication Control Data Structures** for information on the structure.

For a positive response (**req->resp_err = CNF_RESP_OK**), the request control structure member **resp_info_ptr** is a pointer to the operation specific data structure (**JINIT_RESP_INFO**) for the Initiate Journal function.

For a negative response, or other errors, see **Module 11 — MMS-EASE Error Handling** starting on page 3-339 for more information.

Return Value: `ST_VOID` (ignored)

Operation-Specific Data Structure Used: **JINIT_RESP_INFO**

See page 3-218 for a detailed description of this structure.

37. ReportJournalStatus Service

This service is used to determine the number of entries in a Journal object.

Primitive Level ReportJournalStatus Operations

The following section contains information on how to use the paired primitive interface for the ReportJournal-Status service. It covers available data structures used by the PPI, and the four primitive level functions that together make up the ReportJournalStatus service. See **Volume 1 — Module 2 — General Sequence of Events — Confirmed Services** for a detailed explanation of the sequence of events that occur within the MMS-EASE environment during confirmed services.

- The ReportJournalStatus service consists of the paired primitive functions of **mp_jstat**, **u_jstat_ind**, **mp_jstat_resp**, and **u_mp_jstat_conf**.

Data Structures

Request/Indication

The operation-specific structure described below is used by the client in issuing a ReportJournalStatus request (**mp_jstat**). It is received by the server when a ReportJournalStatus indication (**u_jstat_ind**) is received.

```
struct jstat_req_info
  {
  OBJECT_NAME jou_name;
  };
typedef struct jstat_req_info JSTAT_REQ_INFO;
```

Fields:

`jou_name` This structure of type **OBJECT_NAME** contains the name of the Journal for which the status is to be obtained. See **Volume 1 — Module 2 — MMS Object Name Structure** for a detailed description of this structure.

Response/Confirm

The operation-specific data structure described below is used by the server in issuing a ReportJournalStatus response (**mp_jstat_resp**). It is received by the client when a ReportJournalStatus confirm (**u_mp_jstat_conf**) is received.

```
struct jstat_resp_info
  {
  ST_UINT32   cur_entries;
  ST_BOOLEAN  mms_deletable;
  SD_END_STRUCT
  };
typedef struct jstat_resp_info JSTAT_RESP_INFO;
```

Fields:

`cur_entries` This indicates the number of current Journal Entries in this Journal.

`mms_deletable` **SD_FALSE**. This Journal is NOT deletable using a service request.

SD_TRUE. This Journal is deletable using a service request.

Paired Primitive Interface Functions

mp_jstat

Usage: This primitive request function sends a ReportJournalStatus request PDU to a remote node. It uses the data found in a structure of type **JSTAT_REQ_INFO**, pointed to by **info**. This service is used to obtain the status of a Journal at the remote node.

Function Prototype:

```
MMSREQ_PEND *mp_jstat (ST_INT chan,
                       JSTAT_REQ_INFO *info);
```

Parameters:

chan	This is the channel number over which the PDU is to be sent.
info	This pointer to the Operation-Specific data structure of type **JSTAT_REQ_INFO** contains data specific to the request PDU to be sent.

Return Value:

MMSREQ_PEND *	This pointer to the request control data structure of type **MMSREQ_PEND** is used to send the PDU. In case of an error, the pointer is set to null and **mms_op_err** is written with the error code.

Corresponding User Confirmation Function: **u_mp_jstat_conf**

Operation-Specific Data Structure Used: **JSTAT_REQ_INFO**

See page 3-223 for a detailed description of this structure.

u_jstat_ind

Usage: This user indication function is called when a ReportJournalStatus indication is received by a server node. The contents and operation of this function are user-defined, and depend on the application requirements. To conform to the MMS protocol requirements, the application must do one of the following some time after the indication is received:

1) send a positive response using the primitive response function (**`mp_jstat_resp`**) after the specified Journal status has been successfully obtained, or

2) call the appropriate primitive negative (error) response function (**`mp_err_resp`**). See **Module 11 — MMS-EASE Error Handling** on page 3-343 for an explanation of this function.

See **Volume 1 — Module 1 — User Indication Function Class** for additional information regarding the recommended handling of indications.

Function Prototype:

```
ST_VOID u_jstat_ind  (MMSREQ_IND *ind);
```

Parameters:

`ind` This pointer to the **`MMSREQ_IND`** structure is described in **Volume 1 — Module 2 — Request and Indication Control Data Structures**. This pointer must be used to send the response to this indication. The primary item of interest in this structure is **`ind->req_info_ptr`**. This is a pointer to the operation-specific data structure for the ReportJournalStatus indication (**`JSTAT_REQ_INFO`**). This pointer will always be valid when **`u_jstat_ind`** is called.

Return Value: `ST_VOID` (ignored)

Operation-Specific Data Structure Used: **`JSTAT_REQ_INFO`**

See page 3-223 for a detailed description of this structure.

mp_jstat_resp

Usage: This primitive response function sends a ReportJournalStatus positive response PDU using the data from a structure of type **jstat_resp_info**, pointed to by **info**. This function should be called as a response to the **u_jstat_ind** function being called (a ReportJournalStatus indication is received), and after the specified Journal status has been successfully obtained.

Function Prototype:

```
ST_RET mp_jstat_resp (MMSREQ_IND *ind,        JSTAT_RESP_INFO
                                  *info);
```

Parameters:

`ind` This pointer to the indication control structure of type **MMSREQ_IND** is received in the **u_jstat_ind** function.

`info` This pointer to an Operation-Specific data structure of type **JSTAT_RESP_INFO** contains information specific to the response PDU to be sent.

Return Value: `ST_RET` **SD_SUCCESS**. No Error.

`<> 0` Error Code.

Corresponding User Indication Function: **u_jstat_ind**

Operation-Specific Data Structure Used: **JSTAT_RESP_INFO**

See page 3-223 for a detailed description of this structure.

u_mp_jstat_conf

Usage: This primitive user confirmation function is called when a confirm to a ReportJournalStatus request (**mp_jstat**) is received. **resp_err** contains a value indicating whether an error occurred. **resp_info_ptr** may contain data that the remote node sent back in the confirmation or it may contain error information if an error occurred. If no error has occurred (**req->resp_err = CNF_RESP_OK**), ReportJournalStatus information is available using the **req->resp_info_ptr** pointer. See **Volume 1 — Module 1 — Primitive User Confirmation Function Class** for specific information on the recommended handling of confirms.

Note that the function called for this event may be changed if desired; however, **u_mp_jstat_conf** is the standard default function.

Function Prototype:

```
ST_VOID u_mp_jstat_conf (MMSREQ_PEND *req);
```

Parameters:

`req` his pointer to the **MMSREQ_PEND** structure is returned from the original **mp_jstat** request. See **Volume 1 — Module 2 — Request and Indication Control Data Structures** for information on the structure.

For a positive response (**req->resp_err = CNF_RESP_OK**), the request control structure member **resp_info_ptr** is a pointer to the operation specific data structure (**JSTAT_RESP_INFO**) for the ReportJournalStatus function.

For a negative response, or other errors, see **Module 11 — MMS-EASE Error Handling** starting on page 3-339 for more information.

Return Value: `ST_VOID` (ignored)

Operation-Specific Data Structure Used: **JSTAT_RESP_INFO**

See page 3-223 for a detailed description of this structure.

38. CreateJournal Service

This service is used to create a Journal object at a remote node.

Primitive Level CreateJournal Operations

The following section contains information on how to use the paired primitive interface for the CreateJournal service. It covers available data structures used by the PPI, and the four primitive level functions that together make up the CreateJournal service. See **Volume 1 — Module 2 — General Sequence of Events — Confirmed Services** for a detailed explanation of the sequence of events that occur within the MMS-EASE environment during confirmed services.

- The CreateJournal service consists of the paired primitive functions of **`mp_jcreate`**, **`u_jcreate_ind`**, **`mp_jcreate_resp`**, and **`u_mp_jcreate_conf`**.

Data Structures

Request/Indication

The operation-specific structure described below is used by the client in issuing a CreateJournal request (**`mp_jcreate`**). It is received by the server when a CreateJournal indication (**`u_jcreate_ind`**) is received.

```
struct jcreate_req_info
  {
  OBJECT_NAME  jou_name;
  };
typedef struct jcreate_req_info  JCREATE_REQ_INFO;
```

Fields:

`jou_name` This structure of type **`OBJECT_NAME`** contains the name of the Journal to be created. See **Volume 1 — Module 2 — MMS Object Name Structure** for a detailed description of this structure.

Paired Primitive Interface Functions

mp_jcreate

Usage: This primitive request function sends a CreateJournal request PDU to a remote node. It uses the data found in a structure of type **JCREATE_REQ_INFO**, pointed to by **info**. This service is used to create a Journal object at the remote node.

Function Prototype:

```
MMSREQ_PEND *mp_jcreate (ST_INT chan,
                         JCREATE_REQ_INFO *info);
```

Parameters:

chan	This is the channel number over which the PDU is to be sent.
info	This pointer to the Operation-Specific data structure of type **JCREATE_REQ_INFO** contains data specific to the request PDU to be sent.

Return Value:

MMSREQ_PEND *	This pointer to the request control data structure of type **MMSREQ_PEND** is used to send the PDU. In case of an error, the pointer is set to null and **mms_op_err** is written with the error code.

Corresponding User Confirmation Function: **u_mp_jcreate_conf**

Operation-Specific Data Structure Used: **JCREATE_REQ_INFO**

See page 3-229 for a detailed description of this structure.

u_jcreate_ind

Usage: This user indication function is called when a CreateJournal indication is received by a server node. The contents and operation of this function are user-defined, and depend on the application requirements. To conform to the MMS protocol requirements, the application must do one of the following some time after the indication is received:

1) send a positive response using the primitive response function (**`mp_jcreate_resp`**) the specified Journal object have been successfully created, or

2) call the appropriate primitive negative (error) response function (**`mp_err_resp`**). See **Module 11 — MMS-EASE Error Handling** on page 3-343 for an explanation of this function.

See **Volume 1 — Module 1 — User Indication Function Class** for additional information regarding the recommended handling of indications.

Function Prototype: `ST_VOID u_jcreate_ind (MMSREQ_IND *ind);`

Parameters:

`ind` This pointer to the **`MMSREQ_IND`** structure is described in **Volume 1 — Module 2 — Request and Indication Control Data Structures**. This pointer must be used to send the response to this indication. The primary item of interest in this structure is **`ind->req_info_ptr`**. This is a pointer to the operation-specific data structure for the CreateJournal indication (**`JCREATE_REQ_INFO`**). This pointer will always be valid when **`u_jcreate_ind`** is called.

Return Value: `ST_VOID` (ignored)

Operation-Specific Data Structure Used: **`JCREATE_REQ_INFO`**

See page 3-229 for a detailed description of this structure.

mp_jcreate_resp

Usage: This primitive response function sends a CreateJournal positive response PDU. This function should be called as a response to the **`u_jcreate_ind`** function being called (a CreateJournal indication is received), and after the specified Journal object have been successfully created.

Function Prototype: `ST_RET mp_jcreate_resp (MMSREQ_IND *ind);`

Parameters:

`ind` This pointer to the indication control structure of type **`MMSREQ_IND`** is received in the **`u_jcreate_ind`** function.

Return Value: `ST_RET` **`SD_SUCCESS`**. No Error.

`<> 0` Error Code.

Corresponding User Indication Function: **`u_jcreate_ind`**

Operation-Specific Data Structure Used: NONE

u_mp_jcreate_conf

Usage: This primitive user confirmation function is called when a confirm to a CreateJournal request (**mp_jcreate**) is received. **resp_err** contains a value indicating whether an error occurred. **resp_info_ptr** may contain data that the remote node sent back in the confirmation or it may contain error information if an error occurred. See **Volume 1 — Module 1 — Primitive User Confirmation Function Class** for specific information on the recommended handling of confirms.

Note that the function called for this event may be changed if desired; however, **u_mp_jcreate_conf** is the standard default function.

Function Prototype:

```
ST_VOID u_mp_jcreate_conf (MMSREQ_PEND *req);
```

Parameters:

`req` This pointer to the **MMSREQ_PEND** structure is returned from the original **mp_jcreate** request. See **Volume 1 — Module 2 — Request and Indication Control Data Structures** for information on the structure.

For a positive response (**req->resp_err = CNF_RESP_OK**), the request control structure member **resp_info_pres** indicates that there is no confirm data for this function.

For a negative response, or other errors, see **Module 11 — MMS-EASE Error Handling** starting on page 3-339 for more information.

Return Value: `ST_VOID` (ignored)

Operation-Specific Data Structure Used: NONE

39. DeleteJournal Service

This service is used to delete a Journal object at a remote node. Only journal objects having a MMS Deletable attribute equal to TRUE can be deleted using this service.

Primitive Level DeleteJournal Operations

The following section contains information on how to use the paired primitive interface for the DeleteJournal service. It covers available data structures used by the PPI, and the four primitive level functions that together make up the DeleteJournal service. See **Volume 1 — Module 2 — General Sequence of Events — Confirmed Services** for a detailed explanation of the sequence of events that occur within the MMS-EASE environment during confirmed services.

- The DeleteJournal service consists of the paired primitive functions of **`mp_jdelete`**, **`u_jdelete_ind`**, **`mp_jdelete_resp`**, and **`u_mp_jdelete_conf`**.

Data Structures

Request/Indication

The operation-specific structure described below is used by the client in issuing a DeleteJournal request (**`mp_jdelete`**). It is received by the server when a DeleteJournal indication (**`u_jdelete_ind`**) is received.

```
struct jdelete_req_info
  {
  OBJECT_NAME  jou_name;
  };
typedef struct jdelete_req_info  JDELETE_REQ_INFO;
```

Fields:

`jou_name` This structure of type **`OBJECT_NAME`** contains the name of the Journal to be deleted. See **Volume 1 — Module 2 — MMS Object Name Structure** for a detailed description of this structure.

Paired Primitive Interface Functions

mp_jdelete

Usage: This primitive request function sends a DeleteJournal request PDU to a remote node. It uses the data found in a structure of type **JDELETE_REQ_INFO**, pointed to by **info**. This service is used to delete a Journal object at the remote node.

Function Prototype:
```
MMSREQ_PEND *mp_jdelete (ST_INT chan,
                         JDELETE_REQ_INFO *info);
```

Parameters:

chan	This is the channel number over which the PDU is to be sent.
info	This pointer to the Operation-Specific data structure of type **JDELETE_REQ_INFO** contains data specific to the request PDU to be sent.

Return Value:

MMSREQ_PEND *	This pointer to the request control data structure of type **MMSREQ_PEND** is used to send the PDU. In case of an error, the pointer is set to null and **mms_op_err** is written with the error code.

Corresponding User Confirmation Function: **u_mp_jdelete_conf**

Operation-Specific Data Structure Used: **JDELETE_REQ_INFO**

See page 3-235 for a detailed description of this structure.

u_jdelete_ind

Usage: This user indication function is called when a DeleteJournal indication is received by a server node. The contents and operation of this function are user-defined, and depend on the application requirements. To conform to the MMS protocol requirements, the application must do one of the following some time after the indication is received:

1) send a positive response using the primitive response function (**`mp_jdelete_resp`**) after the specified Journal object have been successfully deleted, or

2) call the appropriate primitive negative (error) response function (**`mp_err_resp`**). See **Module 11 — MMS-EASE Error Handling** on page 3-343 for an explanation of this function.

See **Volume 1 — Module 1 — User Indication Function Class** for additional information regarding the recommended handling of indications.

Function Prototype:

```
ST_VOID u_jdelete_ind (MMSREQ_IND *ind);
```

Parameters:

`ind` This pointer to the **`MMSREQ_IND`** structure is described in **Volume 1 — Module 2 — Request and Indication Control Data Structures**. This pointer must be used to send the response to this indication. The primary item of interest in this structure is **`ind->req_info_ptr`**. This is a pointer to the operation-specific data structure for the DeleteJournal indication (**`JDELETE_REQ_INFO`**). This pointer will always be valid when **`u_jdelete_ind`** is called.

Return Value: `ST_VOID` (ignored)

Operation-Specific Data Structure Used: **`JDELETE_REQ_INFO`**

See page 3-235 for a detailed description of this structure.

mp_jdelete_resp

Usage: This primitive response function sends a DeleteJournal positive response PDU. This function should be called as a response to the **`u_jdelete_ind`** function being called (a DeleteJournal indication is received), and after the specified Journal object have been successfully deleted.

Function Prototype: `ST_RET mp_jdelete_resp (MMSREQ_IND *ind);`

Parameters:

`ind` This pointer to the indication control structure of type **`MMSREQ_IND`** is received in the **`u_jdelete_ind`** function.

Return Value: `ST_RET` **`SD_SUCCESS`**. No Error.

`<> 0` Error Code.

Corresponding User Indication Function: **`u_jdelete_ind`**

Operation-Specific Data Structure Used: NONE

u_mp_jdelete_conf

Usage: This primitive user confirmation function is called when a confirm to a DeleteJournal request (**mp_jdelete**) is received. **resp_err** contains a value indicating whether an error occurred. **resp_info_ptr** may contain data that the remote node sent back in the confirmation or it may contain error information if an error occurred. See **Volume 1 — Module 1 — Primitive User Confirmation Function Class** for specific information on the recommended handling of confirms.

Note that the function called for this event may be changed if desired; however, **u_mp_jdelete_conf** is the standard default function.

Function Prototype:

```
ST_VOID u_mp_jdelete_conf (MMSREQ_PEND *req);
```

Parameters:

`req` This pointer to the **MMSREQ_PEND** structure is returned from the original **mp_jdelete** request. See **Volume 1 — Module 2 — Request and Indication Control Data Structures** for information on the structure.

For a positive response (**req->resp_err = CNF_RESP_OK**), the request control structure member **resp_info_pres** indicates that there is no confirm data for this function.

For a negative response, or other errors, see **Module 11 — MMS-EASE Error Handling** starting on page 3-339 for more information.

Return Value: `ST_VOID` (ignored)

Operation-Specific Data Structure Used: NONE

1. Operator Communication Introduction

MMS provides two services, Input and Output, for communication with an operator station. An *operator station* is a MMS object that can be used to represent character based input and output devices that may be attached to the VMD for the purpose of communicating with an operator local to the VMD. MMS defines three types of operator stations:

- **Entry**. An entry only operator station consists of an input device only. This may be keyboard or perhaps a bar code reader. The input data must be of the type VisibleString, consisting of alpha-numeric characters only.
- **Display**. A display only operator station consists of a character based output display that can display VisibleString data (no graphics or control characters).
- **Entry-Display**. This type of operator station consists of both an entry station and a display station.

Because the operator station is a representation of a physical feature of the VMD, it exists beyond the scope of any domain or application association. Therefore, MMS clients access the operator station by name without scope. MMS allows for any number of operator stations of a given VMD. The two services used by MMS clients to perform operator communications are:

Input MMS clients use this service to obtain a single input string from an input device. The service has the option for displaying a sequence of prompts on the display if the operator station is an entry-display type of operation station. See page 3-243 for more information.

Output This service is used to display a sequence of output strings on the display of the operator station. See page 3-249 for more information.

MMS does not specify any flow control mechanism to manage the input and output operations. Therefore, it may be necessary for the MMS-EASE application program to provide flow control to ensure data integrity.

2. Input Service

This service is used by a client to request data from an Operator Station. The requesting user may provide prompt string(s) to be displayed before obtaining the input data.

Primitive Level Input Operations

The following section contains information on how to use the paired primitive interface for the Input service. It covers available data structures used by the PPI, and the four primitive level functions that together make up the Input service. See **Volume 1 — Module 2 — General Sequence of Events — Confirmed Services** for a detailed explanation of the sequence of events that occur within the MMS-EASE environment during confirmed services.

- The Input service consists of the paired primitive functions of **mp_input**, **u_input_ind**, **mp_input_resp**, and **u_mp_input_conf**.

Data Structures

Request/Indication

The operation specific data structure described below is used by the client in issuing an Input request (**mp_input**). It is received by the server when an Input indication (**u_input_ind**) is received.

```
struct input_req_info
  {
  ST_CHAR     station_name [MAX_IDENT_LEN+1];
  ST_BOOLEAN echo;
  ST_BOOLEAN timeout_pres;
  ST_UINT32  timeout;
  ST_BOOLEAN prompt_pres;
  ST_INT     prompt_count;
/*ST_CHAR     *prompt_data [prompt_count];                        */
  SD_END_STRUCT
  };
typedef struct input_req_info INPUT_REQ_INFO;
```

Fields:

`station_name` This indicates the name of the operator station from which the input is to be obtained. This MMS Identifier cannot be greater than 16 characters.

`echo` **SD_FALSE**. Do not echo the input obtained on the operator station display.

SD_TRUE. Echo the input obtained on the operator station display. This is the default.

`timeout_pres` **SD_FALSE**. Do Not include **timeout** in the PDU.

SD_TRUE. Include **timeout** in the PDU.

`timeout` This indicates the number of seconds to wait for the operator to input the data before sending an error response. If this value is zero, and **timeout** is included in the PDU, the remote node should immediately send back an error response. This should occur unless there is already input data waiting at the operator station. If **timeout** is not included in the PDU, the remote node should wait indefinitely for input.

prompt_pres	**SD_TRUE**. Do not include **prompt_count** and **prompt_data** in the PDU.
	SD_FALSE. Include **prompt_count** and **prompt_data** in the PDU.
prompt_count	This indicates the number of elements in the **prompt_data** array. This array contains the number of prompt strings to display before obtaining the input data.
prompt_data	This array of pointers to the null-terminated strings contains the prompt(s) to display before obtaining the input data.

NOTE: FOR REQUESTS ONLY, when allocating a data structure of type **INPUT_REQ_INFO**, enough memory must be allocated to hold the **prompt_data** pointer array and the data pointed to it. The following C statement can be used to allocate this structure, and the pointers:

```
info = (INPUT_REQ_INFO *) chk_malloc(sizeof (INPUT_REQ_INFO) + (prompt_count *
            (sizeof(ST_CHAR *))));
```

Response/Confirm

The operation specific data structure described below is used by the server in issuing an Input response (**mp_input_resp**). It is received by the client when an Input confirm (**u_mp_input_conf**) is received.

```
struct input_resp_info
  {
  ST_CHAR *input_resp;
  };
typedef struct input_resp_info INPUT_RESP_INFO;
```

Fields:

input_resp	This pointer to the character data (visible string) is obtained from the specified operator console.

Paired Primitive Interface Functions

mp_input

Usage: This primitive request function sends an Input request PDU. It uses the data found in a structure of type **INPUT_REQ_INFO,** pointed to by **info**. The Input service is used to obtain operator input from a specified device at the remote node.

Function Prototype:

```
MMSREQ_PEND *mp_input (ST_INT chan,
                       INPUT_REQ_INFO *info);
```

Parameters:

chan	This integer contains the channel number over which the Input request PDU is to be sent.
info	This pointer to an Operation-Specific data structure of type **INPUT_REQ_INFO** contains information specific to the Input request PDU to be sent.

Return Value:

MMSREQ_PEND *	This pointer to the request control data structure of type **MMSREQ_PEND** is used to send the Input PDU. In case of an error, the pointer is set to null and **mms_op_err** is written with the error code.

Corresponding User Confirmation Function: **u_mp_input_conf**

Operation-Specific Data Structure Used: **INPUT_REQ_INFO**

See page 3-243 for a detailed description of this structure.

u_input_ind

Usage: This user function is called when an Input indication is received by a server node. The contents and operation of this function are user-defined, and depend on the application requirements. To conform to the MMS protocol requirements, the application must do one of the following some time after the indication is received:

1) send a positive response using the primitive response function (**mp_input_resp**) after the requested input is successfully obtained from the specified operators console, or

2) call the appropriate primitive negative (error) response function **(mp_err_resp**). See **Module 11 - MMS-EASE Error Handling** on page 3-343 for more information on this function.

See **Volume 1 — Module 1 — Function Classes — Primitive User Confirmation Function Class** for additional information regarding recommended handling of indications.

Function Prototype:

```
ST_VOID u_input_ind  (MMSREQ_IND *ind);
```

Parameters:

`ind` This pointer to the request control structure **MMSREQ_IND** is described in **Volume 1 — Module 2 — Request and Indication Control Data Structures**. This pointer must be used to send the response to this indication. The primary item of interest in this structure is **ind->resp_info_ptr**. This is a pointer to the operation specific information structure (**INPUT_REQ_INFO**). This pointer will always be valid when **u_input_ind** is called.

Return Value: `ST_VOID` (ignored)

Operation-Specific Data Structure Used: **INPUT_REQ_INFO**

See page 3-243 for a detailed description of this structure.

mp_input_resp

Usage: This primitive response function sends an Input positive response PDU. It uses the data from a structure of type **INPUT_RESP_INFO**, pointed to by **info**. This function should be called as a response to the **u_input_ind** function being called (an Input indication is received), and after the requested input is successfully obtained from the specified operators console.

Function Prototype:

```
ST_RET mp_input_resp (MMSREQ_IND *ind,        INPUT_RESP_INFO
                                  *info);
```

Parameters:

`ind` This pointer to the indication control structure of type **MMSREQ_IND** is received in the indication function, **u_input_ind**.

`info` This pointer to an Operation Specific data structure of type **INPUT_RESP_INFO** contains information specific to the Input response PDU to be sent.

Return Value: `ST_RET` **SD_SUCCESS**. No Error.

`<> 0` Error Code.

Corresponding User Indication Function: **u_input_ind**

Operation-Specific Data Structure Used: **INPUT_RESP_INFO**

See page 3-244 for a detailed description of this structure.

u_mp_input_conf

Usage: This primitive user confirmation function is called when an Input confirm to a **mp_input** is received. **resp_err** contains a value indicating whether an error occurred. **resp_info_ptr** may contain data that the remote node sent back in the confirmation or it may contain error information if an error occurred. If no error has occurred, (**req->resp_err = CNF_RESP_OK**), Input information is available to the application using the **req->resp_info_ptr** pointer. See **Volume 1 — Module 1 — Function Classes — Primitive User Confirmation Function Class** for more information on the recommended handling of confirms.

Note that the function called for this event may be changed if required; however, **u_mp_input_conf** is the standard default function.

Function Prototype: `ST_VOID u_mp_input_conf (MMSREQ_PEND *req);`

Parameters:

`req` This pointer of structure type **MMSREQ_PEND** is returned form the original **mp_input** request. See **Volume 1 — Module 2 — Request and Indication Control Data Structures** for more information on this structure.

For a positive response (**req->resp_err = CNF_RESP_OK**), the request control structure member **resp_info_ptr** is a pointer to the operation specific data structure (**INPUT_RESP_INFO**) for the Input function.

For a negative response, or any other errors, see **Module 11 — MMS-EASE Error Handling** starting on page 3-339 for guidelines on handling the error.

Return Value: `ST_VOID` (ignored)

Operation-Specific Data Structure Used: **INPUT_RESP_INFO**

See page 3-244 for a detailed description of this structure.

3. Output Service

This service is used by a client to display data on an Operator Station.

Primitive Level Output Operations

The following section contains information on how to use the paired primitive interface for the Output service. It covers available data structures used by the PPI and the four primitive level functions that together make up the Output service. See **Volume 1 — Module 2 — General Sequence of Events — Confirmed Services** for a detailed explanation of the sequence of events that occur within the MMS-EASE environment during confirmed services.

- The Output service consists of the paired primitive functions of **`mp_output`**, **`u_output_ind`**, **`mp_output_resp`**, **`u_mp_output_conf`**.

Data Structures

Request/Indication

The operation specific data structure described below is used by the client in issuing an Output request (**`mp_output`**). It is received by the server when an Input indication (**`u_output_ind`**) is received.

```
struct output_req_info
  {
  ST_CHAR station_name [MAX_IDENT_LEN+1];
  ST_INT  data_count;
/*ST_CHAR *output_data [data_count];                        */
  SD_END_STRUCT
  };
typedef struct output_req_info OUTPUT_REQ_INFO;
```

Fields:

`station_name`	This contains the name of device to output messages to.
`data_count`	This indicates the number of pointers in the **`output_data`** array.
`output_data`	This array of pointers points to the message to be displayed on the operator station. Each pointer points to a message that should be displayed on a SINGLE line on the display.

NOTE: FOR REQUESTS ONLY, when allocating a data structure of type **`OUTPUT_REQ_INFO`**, enough memory must be allocated to hold the information for the pointer array contained in **`output_data`**. The following C statement can be used:

```
info = (OUTPUT_REQ_INFO *) chk_malloc(sizeof(OUTPUT_REQ_INFO) + (data_count *
            (sizeof(ST_CHAR *))));
```

Paired Primitive Interface Functions

mp_output

Usage: This primitive request function sends an Output request PDU using the data from a structure of type **OUTPUT_REQ_INFO**, pointed to by **info**. The Output service is used to write a message to an operator display or other output device.

Function Prototype:
```
MMSREQ_PEND *mp_output (ST_INT chan,
                        OUTPUT_REQ_INFO *info);
```

Parameters:

chan	This integer contains the channel number over which the Output PDU is to be sent.
info	This pointer to an operation specific data structure of type **OUTPUT_REQ_INFO** contains information specific to the request PDU to be sent.

Return Value:

MMSREQ_PEND *	This pointer to the request control data structure of type **MMSREQ_PEND** is used to send the Output PDU. In case of an error, the pointer is set to null and **mms_op_err** is written with the error code.

Corresponding User Confirmation Function: **u_mp_output_conf**

Operation-Specific Data Structure Used: **OUTPUT_REQ_INFO**

See page 3-249 for a detailed description of this structure.

u_output_ind

Usage: This user function is called when an Output indication is received by a server node. The contents and operation of this function are completely user-defined, and depend on the application. To conform to the MMS protocol requirements, the application must do one of the following some time after the indication is received:

1) send a positive response using the primitive response function (**`mp_output_resp`**) after the specified output data has been successfully displayed on the operator's console, or

2) call the appropriate primitive negative (error) response function (**`mp_err_resp`**). See **Module 11 — MMS-EASE Error Handling** on page 3-343 for more information on this function.

See **Volume 1 — Module 1 — Function Classes — Primitive User Confirmation Function Class** for additional information regarding recommended handling of indications.

Function Prototype:

```
ST_VOID u_output_ind (MMSREQ_IND *ind);
```

Parameters:

`ind` This pointer to the request control structure, **`MMSREQ_IND`** is described in **Volume 1 — Module 2 — Request and Indication Control Data Structures**. This pointer must be used to send the response to this indication. The primary item of interest in this structure is **`ind->resp_info_ptr`**. This is a pointer to the operation specific information structure (**`OUTPUT_REQ_INFO`**). This pointer will always be valid when **`u_output_ind`** is called.

Return Value: `ST_VOID` (ignored)

Operation-Specific Data Structure Used: **`OUTPUT_REQ_INFO`**

See page 2-249 for a detailed description of this structure.

mp_output_resp

Usage: This primitive response function sends an Output positive response PDU. This should be called as a response to the **u_output_ind** function being called (an Output indication is received), and after the specified output data has been successfully displayed on the operator's console.

Function Prototype: `ST_RET mp_output_resp (MMSREQ_IND *ind);`

Parameters:

`ind` This pointer to the indication control structure of type **MMSREQ_IND** is received in the indication function, **u_output_ind**.

Return Value: `ST_RET` **SD_SUCCESS**. No Error.

`<> 0` Error Code.

Corresponding User Indication Function: **u_output_ind**

Operation-Specific Data Structure Used: NONE

u_mp_output_conf

Usage: This primitive user confirmation function is called when an Output confirm to a **mp_output** is received. **resp_err** contains a value indicating whether an error occurred. **resp_info_ptr** may contain data that the remote node sent back in the confirmation or it may contain error information if an error occurred. If no error has occurred, (**req->resp_err = CNF_RESP_OK**), Output information is available to the application using the **req->resp_info_ptr** pointer. See **Volume 1 — Module 1 — Function Classes — Primitive User Confirmation Function Class** for more information on the recommended handling of confirms.

Note that the function called for this event may be changed if required; however, **u_mp_output_conf** is the standard default function.

Function Prototype:

```
ST_VOID u_mp_output_conf (MMSREQ_PEND *req);
```

Parameters:

`req` This pointer of structure type **MMSREQ_PEND** is returned from the original **mp_output** request. See **Volume 1 — Module 2 — Request and Indication Control Data Structures** for more information on this structure.

For a negative response, or other errors, see **Module 11 — MMS-EASE Error Handling** starting on page 3-339 for guidelines on handling the error.

Return Value: `ST_VOID` (ignored)

Operation Specific Data Structure Used: NONE

1. File Access and Management Introduction

MMS provides a set of simple file transfer services for devices that have a local file store but do not support a full set of file services using some other means. For instance, many robot implementations of MMS use the file services for moving program (domain) files to the robot from a client application. The MMS file services support file transfer only, not file access. Although these file services are defined within an annex within the MMS standard, they are widely supported by most commercial MMS implementations. The services for files are:

File Access

FileOpen This service is used by a client to tell the VMD to open a file and prepare it for a file transfer. See page 3-277 for more information.

FileRead This service is used to obtain a segment of the file's data from a VMD. The client would continue to issue FileOpen requests until the VMD indicates that all the data in the file has been returned. The number of bytes returned in each FileRead response is determined solely by the VMD (and can vary from one FileRead response to the next) and is not controllable by the client. See page 3-287 for more information.

FileClose This service is used by a client to close a previously opened file. It is used after all the data from the file has been read or can beused to discontinue a file transfer before it is completed. See page 3-297 for more information.

ObtainFile This service is used to by a client to tell the VMD to obtain a file. When a VMD receives an ObtainFile request, it will issue a FileOpen, FileRead(s), and FileClose service requests to the client application that issued the ObtainFile request. The client will then have to support the server functions of the FileOpen, FileRead, and FileClose services. A third party option is available (if supported by the VMD) to tell the VMD to obtain the file from another node on the network using some protocol (that may or may not be MMS). See page 3-269 for more information.

FileCopy This is a MMS-EASE virtual machine service consisting of a FileOpen, FileRead, and FileClose. See page 3-305 for more information.

File Management

FileRename This service is used to rename a file on the VMD. See page 3-309 for more information.

FileDelete This service is used to delete a file on the VMD. See page 3-315 for more information.

FileDirectory This service is used to obtain a file directory on the VMD. See page 3-321 for more information.

2. Virtual Machine Interface

The virtual machine can take care of most file operations if those files are resident on the host on which MMS-EASE executes. If using MMS-EASE to create a gateway application, the local file operations may have to be handled independently of MMS-EASE. This would be an application where the files are actually contained on an external system such as a robot. In these applications, it will be necessary to use the MMS-EASE PPI (Paired Primitive Interface) for communications.

The Virtual Machine provides the following services for File Access:

1. Keeps a record of local and remote files opened over the network.
2. Allows opening and closing of remote files by path name and channel.
3. Allows reading of any number of blocks of remote files into a user-specified buffer. The MMS specification only allow one block from the file to be transferred in any given PDU. MMS-EASE will group PDUs automatically to read any number of blocks desired if you use the **`mv_fread`** function to read files.
4. Handles all file access indications without user program intervention, including FileOpen, FileRead, FileClose, FileDelete, FileRename and FileDirectory indications.
5. Allows the user program to configure the maximum number of files that can be opened locally and remotely.
6. Allows the user to copy files using one function call to **`mv_fcopy`**.
7. Automatically obtains files when the ObtainFile indication specifies that the file to be obtained exists on the same node issuing the ObtainFile request.

File Management Structures

The following structures, **`FCTRL_INFO`, `FCTRLINFO`**, **`LOCAL_FCTRL_INFO`**, and **`REM_FCTRL_INFO`** are the file control structures. These are used by MMS-EASE to keep track of files both remote and local. All these structures are allocated during powerup (**`strt_MMS`**) depending on the values of **`max_loc_open`**, **`max_rem_open`**, and **`MAXFILESOPEN`**. See **Volume 1 — Module 2 — MMS Configuration Variables** for more information.

NOTES:

1. NONE OF THE PARAMETERS WITHIN THESE STRUCTURES SHOULD EVER BE WRITTEN TO BY YOUR PROGRAM. They are included here for information purposes only.
2. MMS specifies the use of graphics strings for file and path names. Therefore, field lengths for file names must be written even if using a standard string manipulation library function to write a pure ASCII file name into a character array.

File Control Information Structures

fctrl_info

This structure contains information about a particular file accessed by the virtual machine. The state of the various files can be determined by using the support functions **ms_get_rem_finfo** and **ms_get_loc_finfo**. These both return a pointer to a structure of this type. See page 3-264 for more information on these support functions. The members of this structure should only be read (not written) by the user.

```
struct fctrl_info
  {
  ST_INT    refnum;
  ST_INT    chan;
  ST_INT32  frsmid;
  FILE      *handle;
  ST_UINT32 fsize;
  time_t    open_time;
  time_t    acc_time;
  ST_INT    path_len;
  ST_CHAR    path[MAX_FILE_NAME];
  ST_INT    dest_path_len;
  ST_CHAR    dest_path[MAX_FILE_NAME];
  ST_UCHAR   *buffer;
  ST_LONG    bytes_read;
  ST_LONG    bytes_req;
  };
typedef struct fctrl_info FCTRL_INFO;
```

Fields:

`refnum` This reference number is the index into the **FCTRLINFO** structure. See the following pages for further information on this structure.

`chan` This is the channel number assigned to the associations that requested the file open.

`frsmid` This is the **F**ile **R**ead **S**tate **M**achine **ID**. This ID is used to keep track of a instance of a file transfer (open, read, close).

`handle` This is an Operating System pointer to the file in local memory.

`fsize` This is the length, in bytes, of the file.

`open_time` This is the local machine time the file was opened.

`acc_time` This is the local time that the file was last accessed.

`path_len` This is the length, in bytes, of the file and path name.

`path` This is an array of characters containing the file and path names.

`dest_path_len` This is the length, in bytes, of the destination file and path.

`dest_path` This is an array of integers containing the destination file and path names.

`buffer` This is a pointer to the buffer containing the destination of the file data.

`bytes_read` This contains the number of bytes read into buffer so far.

`bytes_req` This contains the number of bytes requested to be read.

fctrlinfo

This structure contains status information, and pointers to the **FCTRL_INFO** structures for files accessed by the virtual machine.

```
struct fctrlinfo
  {
  ST_CHAR    stat[MAX_FILES_OPEN];
  FCTRL_INFO *finfo[MAX_FILES_OPEN];
  };
typedef struct fctrlinfo FCTRLINFO;
```

Fields:

`stat` This indicates the file status:

- **0** **MFSTAT_NO_ACT** (No Activity)
- **1** **MFSTAT_PEND_OPEN** (Pending Open)
- **2** **MFSTAT_OPENED** (Open)
- **3** **MFSTAT_PEND_CLOSE** (Pending Close)

`finfo` This pointer of structure type **FCTRL_INFO** contains the file control information. The default (**MAX_FILES_OPEN**) is set to 20.

MMS-EASE provides two structures, **m_local_fctrl_info** and **m_rem_fctrl_info**, of the structure type **FCTRLINFO**. These contain information about the local and remote files, respectively, that the virtual machine has been used to access.

```
extern FCTRLINFO m_local_fctrl_info;     /* local file control info        */

extern FCTRLINFO m_rem_fctrl_info;               /* remote file control info      */
```

To determine the status of a particular file, either **ms_get_loc_finfo** (for local files), or **ms_get_rem_finfo** (for remote files) must be called first. These functions return a pointer to the appropriate **FCTRL_INFO** structure for the specified file. Then, you can use the **refnum** field of the **FCTRL_INFO** structure with the **m_local_fctrl_info** or **m_rem_fctrl_info** structures to determine the status (e.g., **m_local_fctrl_info[refnum]** or **m_rem_fctrl_info[refnum]**).

NOTES:

1. Your application programs are responsible for insuring that certain file operations do not exceed the maximum message sizes negotiated when associations are established. This parameter is **mms_chan_info[chan].file_blk_size**. This is calculated at the time of association negotiation by the virtual machine in **mv_init** or **mv_init_resp**. See **Volume 1 — Module 3 — Initiate Service** for more information.

2. The virtual machine makes use of a simplified means of representing file names that differs from the way MMS represents file names. The MMS standard represents file names by a sequence of graphics strings (a character string that can contain any type of textual and graphic data). The MMS-EASE virtual machine automatically combines the sequence of file names into a single file name of a maximum length of **MAX_FILE_NAME**. This occurs before it is stored in the data structures described above.

File Control Global Variables

The following variables indicate the number of files currently open (locally or remotely) and the user-defined maximum number for files accessed using the virtual machine. These variables should be changed before strt_MMS is called.

`extern ST_INT m_rem_files_open;` This variable indicates the number of remote files open.

`extern ST_INT m_loc_files_open;` This variable indicates the number of local files open.

`extern ST_INT max_loc_open;` This variable indicates a user-defined maximum number of open remote files. The default (**`MAX_FILES_OPEN`**) is set to 20.

`extern ST_INT max_rem_open;` This variable indicates a user-defined maximum number of open local files. The default (**`MAX_FILES_OPEN`**) is set to 20.

Pipelined File Transfer Capability

An application can request that pipelining be used in the FileCopy Request and ObtainFile Response virtual machine functions. Pipelining file read requests can improve file transfer performance significantly. To choose the pipeline option, a global channel-oriented array of integers is used that controls the pipeline depth. This specifies the pipeline depth for file copies on a channel by channel basis.

NOTE: This is an allocated array, and cannot be accessed until AFTER **`strt_MMS`** is called.

```
extern ST_INT *m_fcopy_pipe_depth;
```

Values of 0 and 1 are both equivalent and cause no pipelining to occur. The maximum pipeline depth allowed will be the constant **`MAX_FCOPY_PIPELINE_DEPTH`**. This value is currently defaulted to 10.

The pipe depth is that which is in **`m_fcopy_pipe_depth[chan]`**at the time of the start of the operation (File-Copy Request or ObtainFile response). It will not be changed during the transfer even if the value in **`m_fcopy_pipe_depth[chan]`** is changed.

NOTE: If the virtual machine cannot send a file read request (due to other channel traffic or other reasons), the error will be ignored if it is a pipelined request.

3. File Support Functions

These functions are called by the user's application program to perform the various support functions required by MMS-EASE to manage File services. For File Operations, they are used to reset the VMI database for local and remote files. The contents of these functions are determined by MMS-EASE.

ms_clr_locl_fctrl

Usage: This support function can be used during the processing of Conclude indications or when terminating associations. It resets the virtual machine's database concerning local file status and activity. It closes all the local files opened over this channel. This function should only be called when no more activity on the requested channel will occur for the current association. It also should be called if the association is no longer active and the possibility of a corrupt virtual machine database exists (possibly due to the association being aborted during the middle of file operations).

Function Prototype: `ST_RET ms_clr_locl_fctrl (ST_INT chan);`

Parameters:

`chan` This is the channel number for which to reset the virtual machine database concerning local file status and activity. If equal to **-1**, this function will reset the local file data for all channels.

Return: `ST_RET` **`SD_SUCCESS`**. No Error.

`<> 0` Error Code.

Data Structure Used: **`m_local_fctrl_info`**

See page 3-260 for more information.

NOTE: This function should only be called when the association over which the local file activity occurred is no longer active. If this function is called when the association is still active and there is additional file activity performed using the virtual machine, the results will be indeterminate.

ms_clr_rem_fctrl

Usage: This support function can be used during the processing of Conclude indications or when terminating associations. It resets the virtual machine's database concerning remote file status and activity. This function should only be called when no more activity on the requested channel will occur for the current association. It also should be called if the association is no longer active and the possibility of a corrupt virtual machine database exists (possibly due to the association being aborted during the middle of file operations).

Function Prototype: `ST_RET ms_clr_rem_fctrl (ST_INT chan);`

Parameters:

`chan` This is the channel number for which to reset the virtual machine database concerning remote file status and activity. If equal to **-1**, this function will reset the remote file data for all channels.

Return: `ST_RET` **`SD_SUCCESS`**. No Error

`<> 0` Error

Data Structure Used: **`m_rem_fctrl_info`**

See page 3-260 for more information.

NOTE: This function should only be called when the association over which the remote file activity occurred is no longer active. If this function is called when the association is still active and there is additional file activity performed via the virtual machine, the results will be indeterminate.

ms_fname_cat

Usage: This function is used to catenate the contents of an array of **FILE_NAME** structures into a null-terminated Visible string.

Function Prototype:

```
ST_INT ms_fname_cat (FILE_NAME *fn,
                     ST_INT num_fn,
                     ST_CHAR *dest,
                     ST_INT dest_len);
```

Parameters:

`fn` This is a pointer to the beginning of an array of structures of type **FILE_NAME**. Each element in the array contains a portion of the file name. See page 3-267 for more information on this structure.

`num_fn` This indicates the number of elements in the **fn** array.

`dest` This points to the array in the application which will contain the catenated FileName.

`dest_len` This is the length of the array supplied by the application. The function will return rather than write filename contents beyond this element in the array

Return Value:

`ST_INT` This function returns the length of the catenated FileName.

ms_get_loc_finfo

Usage: This function returns a pointer to the file control structure for the local file with the specified path name.

Function Prototype: `FCTRL_INFO *ms_get_loc_finfo (ST_CHAR *fname);`

Parameters:

`fname` This pointer to the local file name specifies the file opened as a result of an incoming FileOpen indication using **`mv_fopen_resp`**. See page 3-284 for more information on this function.

Return Value:

`FCTRL_INFO *` This is a pointer to the file control structure for the local path name specified. In case of an error, the pointer is set to null and **`mms_op_err`** is written with the error code.

Data Structure Used: **`FCTRL_INFO`**

See page 3-258 for more information on this structure.

ms_get_rem_finfo

Usage: This function returns a pointer to the file control structure for the remote file with the specified path name.

Function Prototype:

```
FCTRL_INFO *ms_get_rem_finfo (ST_CHAR *fname,
                              ST_INT chan);
```

Parameters:

`fname`	This pointer to the remote file name specifies the file opened as a result of an incoming FileOpen request using **`mv_fopen`**. See page 3-283 for more information on this function.
`chan`	This integer contains the channel number over which the FileOpen request was sent.

Return Value:

`FCTRL_INFO *`	This is a pointer to the file control structure for the remote path name specified. In case of an error, the pointer is set to null and **`mms_op_err`** is written with the error code.

Data Structure Used: **`FCTRL_INFO`**

See page 3-258 for more information on this structure.

4. Paired Primitive Interface

File Operation Specific Support Structure

The following structure is used to represent MMS file names. Because MMS represents file names as sequences of graphics strings, this structure is used as an element in array of these structures. The number of elements in the array is equal to the number of elements comprising the file name sequence. This structure is used by all the MMS services supported using the PPI containing file names. The actual form and content of a file name is not defined by the MMS standard. It is determined by each application and the operating system requirements. Most systems only require one file name to be specified containing both the path and file name information in one string. Most file names are visible strings as visible strings are a subset of graphic strings.

```
struct file_name
  {
  ST_INT  fn_len;
  ST_CHAR *fname;
  SD_END_STRUCT
  };
typedef struct file_name FILE_NAME;
```

Fields:

`fn_len`	This is the length in bytes of the file name string.
`fname`	This pointer to the graphic string contains at least one element of the file name sequence.

5. ObtainFile Service

This service allows a client to request a server to obtain a specified file from the client's designated filestore, or from a third party's filestore. The responding server is not obligated to use the MMS services to perform the file transfer.

Primitive Level ObtainFile Operations

The following section contains information on how to use the paired primitive interface for the ObtainFile service. It covers available data structures used by the PPI, and the four primitive level functions that together make up the ObtainFile service. See **Volume 1 — Module 2 — General Sequence of Events — Confirmed Services** for a detailed explanation of the sequence of events that occur within the MMS-EASE environment during confirmed services.

- The ObtainFile service consists of the paired primitive functions of **`mp_obtfile`**, **`u_obtfile_ind`**, **`mp_obtfile_resp`**, and **`u_mp_obtfile_conf`**.

Data Structures

Request/Indication

The operation specific data structure described below is used by the client in issuing an ObtainFile request (**`mp_obtfile`**). It is received by the server when an ObtainFile indication (**`u_obtfile_ind`**) is received.

```
struct obtfile_req_info
  {
  ST_BOOLEAN ar_title_pres;
  ST_INT     ar_len;
  ST_UCHAR   *ar_title;
  ST_INT     num_of_src_fname;
  ST_INT     num_of_dest_fname;
/*FILE_NAME  src_fname_list[ num_of_src_fname ];          */
/*FILE_NAME  dest_fname_list[ num_of_dest_fname ];        */
  SD_END_STRUCT
  };
typedef struct obtfile_req_info  OBTFILE_REQ_INFO;
```

Fields:

`ar_title_pres` — **`SD_FALSE`**. Do not include **`ar_title`** in the PDU. The file can be obtained from the local node.

`SD_TRUE`. Include **`ar_title`** in the PDU. The file should be obtained from the application process identified by **`ar_title`**.

`ar_len` — This is the length, in bytes, of **`ar_title`**.

`ar_title` — This pointer to an explicit ASN.1 encoded object identifier specifies the AP title of the system from which the file should be obtained.

`num_of_src_fname` — This indicates the number of elements in the file name sequence specifying the name of the source file.

`num_of_dest_fname` — This indicates the number of elements in the file name sequence specifying the name that the file should have after it has been obtained.

`src_fname_list` This list of structures of type **`FILE_NAME`** contains the elements comprising the file name sequence. It indicates the name of the file to be obtained.

`dest_fname_list` This list of structures of type **`FILE_NAME`** contains the elements comprising the file name sequence. It indicates the name the file should have after it was obtained.

NOTES:

1. See the description of the file name structure on page 3-267 for more information on the format of MMS file names.
2. FOR REQUESTS ONLY, when allocating a data structure of type **`OBTFILE_REQ_INFO`**, enough memory must be allocated to hold the information for the file name sequence. The following C statement can be used:

```
info = (OBTFILE_REQ_INFO *) chk_malloc(sizeof(OBTFILE_REQ_INFO) +
            (num_of_src_fname + num_of_dest_fname)* sizeof(FILE_NAME));
```

Paired Primitive Interface Functions

mp_obtfile

Usage: This primitive request function sends an ObtainFile request PDU using the data from a structure of type **OBTFILE_REQ_INFO**, pointed to by **info**. This service allows the local program to request that the remote node obtain a file from the local or another node. This is the method by which files are transferred to a remote node. It is the responsibility of the remote node, or a subordinate file server, to open, read, and close the specified file.

Function Prototype:

```
MMSREQ_PEND *mp_obtfile (ST_INT chan,
                         OBTFILE_REQ_INFO *info);
```

Parameters:

`chan` This integer contains the channel number over which the ObtainFile PDU is to be sent.

`info` This pointer to a data structure of type **OBTFILE_REQ_INFO** contains information specific to the ObtainFile PDU to be sent.

Return Value:

`MMSREQ_PEND *` This pointer to the request control data structure of type **MMSREQ_PEND** is used to send the ObtainFile PDU. In case of an error, the pointer is set to null and **mms_op_err** is written with the error code.

Corresponding User Confirmation Function: **u_mp_obtfile_conf**

Operation-Specific Data Structure Used: **OBTFILE_REQ_INFO**

See page 3-269 for a detailed description of this structure.

u_obtfile_ind

Usage: This user indication function is called when an ObtainFile indication is received by a server node. The contents and operation of this function are user-defined, and depend on the application requirements. To conform to the MMS protocol requirements, the application must do one of the following some time after the indication is received:

1) send a positive response using either:

   a) the primitive response function (**mp_obtfile_resp**) after the file has been successfully obtained, or

   b) the virtual machine response function (**mv_obtfile_resp**) to have the virtual machine take the requested action and send the response, or

2) call the appropriate primitive negative (error) response function (**mp_err_resp**). Please refer to **Module 11 - MMS-EASE Error Handling** on page 3-343 for more information on this function.

See **Volume 1 — Module 1 — User Indication Function Class** for additional information regarding the handling of indications.

Function Prototype: `ST_VOID u_obtfile_ind (MMSREQ_IND *ind);`

Parameters:

`ind` This pointer to the **MMSREQ_IND** structure is described in **Volume 1 — Module 2 — Request and Indication Control Data Structures**. This pointer must be used to send the response to this indication. The primary item of interest in this structure is **ind->resp_info_ptr**. This is a pointer to the operation specific data structure for an ObtainFile indication (**OBTFILE_REQ_INFO**). This pointer will always be valid when **u_obtfile_ind** is called.

Return Value: `ST_VOID` (ignored)

Operation-Specific Data Structure Used: **OBTFILE_REQ_INFO**

See page 3-269 for a detailed description of this structure.

mp_obtfile_resp

Usage: This primitive response function sends an ObtainFile positive response PDU. This function should be called after the **u_obtfile_ind** function is called (an ObtainFile indication is received), and after the file has been successfully obtained.

Function Prototype: `ST_RET mp_obtfile_resp (MMSREQ_IND *ind);`

Parameters:

`ind` This pointer to the indication control structure of type **MMSREQ_IND** is received in the **u_obtfile_ind** function.

Return Value: `ST_RET` **SD_SUCCESS**. No Error.

`<> 0` Error Code.

Corresponding User Indication Function: **u_obtfile_ind**

Operation-Specific Data Structure Used: NONE

u_obtfile_resp_done

Usage: This user function pointer can be used to reference a done function that allows the server to know when it has completed the obtainfile process. Normally a server will respond to an ObtainFile indication by calling **`mv_obtfile_resp`**. A file transfer state machine will start and run to completion without further interaction by the application. When this function pointer is set to a non-null value, it is used as a callback function to the server application.

Function Prototype:

```
extern ST_VOID (*u_obtfile_resp_done)(ST_RET code, MMSREQ_IND
                                                    *ind);
```

Parameters:

`code`	**`SD_SUCCESS`**. This indicates that the server obtained a file successfully
	`SD_FAILURE`. This indicates that the server did not obtain a file successfully
`ind`	This pointer to the indication control structure of type **`MMSREQ_IND`** is received in the **`u_obtfile_ind`** function.

Return Value: `ST_VOID` (ignored)

u_mp_obtfile_conf

Usage: This primitive user confirmation function is called when the confirm to a **mp_obtfile** is received. **resp_err** contains a value indicating whether an error occurred. **resp_info_ptr** may contain data that the remote node sent back in the confirmation or it may contain error information if an error occurred. If no error occurs (**req->resp_err = CNF_RESP_OK**), ObtainFile information is available to the application using the **req->resp_info_ptr** pointer. See **Volume 1 — Module 1 — Primitive User Confirmation Function Class** for more information on the recommended handling of confirms.

Note that the function called for this event may be changed if desired; however, **u_mp_obtfile_conf** is the standard default function.

Function Prototype:

```
ST_VOID u_mp_obtfile_conf (MMSREQ_PEND *req);
```

Parameters:

`req` This pointer to the **MMSREQ_PEND** structure is returned from the original **mp_obtfile** request function. See **Volume 1 — Module 2 — Request and Indication Control Data Structures** for more information on this structure.

For a negative response, or other errors, see **Module 11 — MMS-EASE Error Handling** starting on page 3-339 for more information.

Return Value: `ST_VOID` (ignored)

Operation Specific Data Structure Used: NONE

Virtual Machine ObtainFile Operations

The following section contains information on how to use the virtual machine interface for the ObtainFile service.

- The ObtainFile service consists of the virtual machine function of **`mv_obtfile_resp`**.

There are no virtual machine data structures.

Virtual Machine Interface Functions

mv_obtfile_resp

Usage: This virtual machine function allows the user to respond to an ObtainFile indication without actually having to obtain the remote file directly, and without having to interact with the operating system to obtain the file. This function takes care of all the PDUs and operating system calls necessary to implement the ObtainFile.

Function Prototype: `ST_RET mv_obtfile_resp (MMSREQ_IND *ind);`

Parameters:

`ind` This pointer to a indication control data structure of the type **`MMSREQ_IND`** is received for the ObtainFile indication.

Return Value: `ST_RET` **`SD_SUCCESS`**. No Error.

`<> 0` Error Code.

NOTES:

1. Because the virtual machine takes care of all the details of obtaining the file transparently to the user's program, the application may use the **`u_obtfile_resp_done`** function pointer if it is necessary to know when the file has been obtained.
2. An ObtainFile indication, processed by the virtual machine using this function, cannot be canceled. You may take down the connection over which the file is being transferred to stop the transfer.
3. Pipeline depth can be specified for an Obtain File response. See page 3-260 for more information.

6. FileOpen Service

This service is used to identify a file to be read, and to establish the open state for the **F**ile **R**ead **S**tate **M**achine (**FRSM**). The client specifies the name of the file, and an initial read position.

Paired Primitive Level FileOpen Operations

The following section contains information on how to use the paired primitive interface for the FileOpen service. It covers available data structures used by the PPI, and the four primitive level functions that together make up the FileOpen service. See **Volume 1 — Module 2 — General Sequence of Events — Confirmed Services** for a detailed explanation of the sequence of events that occur within the MMS-EASE environment during confirmed services.

- The FileOpen service consists of the paired primitive functions of **`mp_fopen`**, **`u_fopen_ind`**, **`mp_fopen_resp`**, and **`u_mp_fopen_conf`**.

Data Structures

Request/Indication

The operation specific data structure described below is used by the client in issuing a FileOpen request (**`mp_fopen`**). It is received by the server when a FileOpen indication (**`u_fopen_ind`**) is received.

```
struct fopen_req_info
  {
  ST_INT    num_of_fname;
  ST_UINT32 init_pos;
/*FILE_NAME  fname_list[ num_of_fname ];                    */
  SD_END_STRUCT
  };
typedef struct fopen_req_info FOPEN_REQ_INFO;
```

Fields:

`num_of_fname` This indicates the number of elements in the file name sequence specifying the name of the file to be opened.

`init_pos` This indicates the number of bytes into the file after which the first FileRead will begin to read data. If this value is zero, the first File Read should start at the beginning of the file.

`fname_list` This list of structures of type **`FILE_NAME`** contains the elements that comprise the file name sequence indicating the name of the file to be opened.

NOTES:

1. See the description of the file name structure on page 3-267 for more information on the format of MMS file names.
2. FOR REQUESTS ONLY, when allocating a data structure of type **`FOPEN_REQ_INFO`**, enough memory must be allocated to hold the information for the file name sequence. The following C statement can be used:

```
info = (FOPEN_REQ_INFO *) chk_malloc(sizeof(FOPEN_REQ_INFO) + num_of_fname *
            sizeof(FILE_NAME));
```

Response/Confirm

The operation specific data structures described below are used by the server in issuing a FileOpen response (**mp_fopen**). It is received by the client when a FileOpen confirm (**u_mp_fopen_conf**) is received.

```
struct fopen_resp_info
  {
  ST_INT32  frsmid;
  FILE_ATTR ent;
  };
typedef struct fopen_resp_info FOPEN_RESP_INFO;
```

Fields:

frsmid	This contains the **F**ile **R**ead **S**tate **M**achine **ID** assigned to this file. All future FileRead requests should reference this number.
ent	This structure of type **FILE_ATTR** contains the file attributes for this file. See below for a description of this structure.

AND:

```
struct file_attr
  {
  ST_UINT32  fsize;
  ST_BOOLEAN mtimpres;
  time_t     mtime;
  };
typedef struct file_attr FILE_ATTR;
```

Fields:

fsize	This contains the size of the file, in bytes.
mtimpres	**SD_FALSE**. **mtime** is not included in the PDU. **SD_TRUE**. **mtime** is included in the PDU.
mtime	This contains the time, in the C language format, **time_t**, that the file was last modified.

Paired Primitive Interface Functions

mp_fopen

Usage: This primitive request function sends a FileOpen request PDU using the information contained in a structure of type **FOPEN_REQ_INFO**, pointed to by **info**. The FileOpen service is used to open a file for reading at the remote node.

Function Prototype:

```
MMSREQ_PEND *mp_fopen (ST_INT chan,
                       FOPEN_REQ_INFO *info);
```

Parameters:

chan	This integer contains the channel number over which the FileOpen PDU is to be sent.
info	This pointer to a data structure of type **FOPEN_REQ_INFO** contains information specific to the FileOpen request.

Return Value:

MMSREQ_PEND *	This pointer to the request control data structure of type **MMSREQ_PEND** is used to send the FileOpen PDU. In case of an error, the pointer is set to null and **mms_op_err** is written with the error code.

Corresponding User Confirmation Function: **u_mp_fopen_conf**

Operation-Specific Data Structure Used: **FOPEN_REQ_INFO**

See page 3-277 for a detailed description of this structure.

u_fopen_ind

Usage: This user indication function is called when a FileOpen indication is received by a server node. The contents and operation of this function are user-defined, and depend on the application requirements. To conform to the MMS protocol requirements, the application must do one of the following some time after the indication is received:

1) send a positive response using either:

   a) the primitive response function (**mp_fopen_resp**) after the file is successfully opened, or

   b) the virtual machine response function (**mv_fopen_resp**) to have the virtual machine take the requested action and send the response, or

2) call the appropriate primitive negative (error) response function (**mp_err_resp**). See **Module 11 — MMS-EASE Error Handling** on page 3-343 for more information on this function.

See **Volume 1 — Module 1 — User Indication Function Class** — for more information regarding the recommended handling of indications.

Function Prototype: `ST_VOID u_fopen_ind (MMSREQ_IND *ind);`

Parameters:

`ind` This pointer to the **MMSREQ_IND** structure is described in **Volume 1 — Module 2 — Request and Indication Control Data Structures**. This pointer must used to send the response to this indication. The primary item of interest in this structure is **ind->resp_info_ptr**. This is a pointer to the operation specific data structure for a FileOpen indication (**FOPEN_REQ_INFO**). This pointer will always be valid when (**u_fopen_ind**) is called.

Return Value: `ST_VOID` (ignored)

Operation-Specific Data Structure Used: **FOPEN_REQ_INFO**

See page 3-277 for a detailed description of this structure.

mp_fopen_resp

Usage: This primitive response function sends a FileOpen positive response PDU using the data from a structure of type **FOPEN_RESP_INFO**, pointed to by **info**. This function should be called after the **u_fopen_ind** function is called (a FileOpen indication was received), and after the file is successfully opened.

Function Prototype:

```
ST_RET mp_fopen_resp (MMSREQ_IND *ind,
                      FOPEN_RESP_INFO *info);
```

Parameters:

`ind` This pointer to the indication control structure of type **MMSREQ_IND** is received in the **u_fopen_ind** function.

`info` This pointer to an Operation Specific data structure of type **FOPEN_RESP_INFO** contains information specific to the FileOpen response PDU to be sent.

Return Value: `ST_RET` **SD_SUCCESS**. No Error.

`<> 0` Error Code.

Corresponding User Indication Function: **u_fopen_ind**

Operation-Specific Data Structure Used: **FOPEN_RESP_INFO**

See page 3-278 for a detailed description of this structure.

u_mp_fopen_conf

Usage: This a primitive user confirmation function is called when a confirm to a **mp_fopen** is received. **resp_err** contains a value indicating whether an error occurred. **resp_info_ptr** may contain data that the remote node sent back in the confirmation or it may contain error information if an error occurred. If no error has occurred (**req->resp_err = CNF_RESP_OK**), FileOpen information is available to the application using the **req->resp_info_ptr** pointer. See **Volume 1 — Module 1 — Primitive User Confirmation Function Class** for more information on the recommended handling of confirms.

Note that the function called for this event may be changed if desired; however, **u_mp_fopen_conf** is the standard default function.

Function Prototype: `ST_VOID u_mp_fopen_conf (MMSREQ_PEND *req);`

Parameters:

`req` This pointer to the **MMSREQ_PEND** structure is returned from the original **mp_fopen** request function. See **Volume 1 — Module 2 — Request and Indication Control Data Structures** for more information on this structure.

For a positive response (**req->resp_err = CNF_RESP_OK**), the request control structure member **resp_info_ptr** is a pointer to the operation specific data structure (**FOPEN_RESP_INFO**) for the FileOpen function.

For a negative response, or other errors, see **Module 11 — MMS-EASE Error Handling** starting on page 3-339 for more information.

Return Value: `ST_VOID` (ignored)

Operation-Specific Data Structure Used: **FOPEN_RESP_INFO**

See page 3-278 for a detailed description of this structure.

Virtual Machine FileOpen Operations

The following section information on how to use the virtual machine interface for the FileOpen service. It covers available data structures used by the VMI, and the virtual machine functions that together make up the File Open service.

- The FileOpen service consists of the virtual machine functions of **mv_fopen**, **mv_fopen_resp**, and **u_mv_fopen_conf**.

In the virtual machine confirm function, **u_mv_fopen_conf**, it uses the **FOPEN_RESP_INFO** and **FILE_ATTR** data structures. See page 278 for more information on these structures.

Virtual Machine Interface Functions

mv_fopen

Usage: This virtual machine function allows the user to execute the FileOpen service without having to create an input structure of type **FOPEN_REQ_INFO** explicitly. The necessary state information regarding this file transfer is initialized. The user may want to access the outputs from the FileOpen service. These consist of such file directory information as the size of the file to be read, content type, and File Read State Machine ID (FRSMID). This information can be accessed from the user-defined function **u_mv_fopen_conf**. It is found in a structure, of type **FCTRL_INFO**. This is pointed to by **req->resp_info_ptr** where **req** is the input argument to **u_mv_fopen_conf**.

Function Prototype:

```
MMSREQ_PEND *mv_fopen  (ST_INT chan,
                        ST_CHAR *path,
                        ST_INT path_len,
                        ST_UINT32 init_pos);
```

Parameters:

chan	This indicates the channel on which file transfer is to take place.
path	This contains the path and file name of the remote file to be opened.
path_len	This contains the length, in bytes, of the file and path name.
init_pos	This indicates the initial position in the file from which subsequent File Reads should take place. This should be equal to the number of bytes from the beginning of the file from which the next FileRead will start.

Return Value:

MMSREQ_PEND *	This pointer to a request control data structure of type **MMSREQ_PEND** holds information regarding this request. In case of an error, the pointer is set to null and **mms_op_err** is written with the error code.

Corresponding Confirmation Function: **u_mv_fopen_conf**

mv_fopen_resp

Usage: This virtual machine function allows the user to respond to a FileOpen indication (**u_fopen_ind**) without having to create an input structure of type **FOPEN_RESP_INFO** ahead of time. The requested file is opened, state information is initialized for the ensuing file transfer, and a FileOpen response is sent back to the requesting MMS-user. Note that if this virtual machine function is used to respond to a FileOpen indication, the other virtual machine functions, **mv_fread_resp** and **mv_fclose_resp** should also be used, rather than the paired primitive functions, **mp_fread_resp** and **mp_fclose_resp**.

Function Prototype:

```
ST_RET mv_fopen_resp (MMSREQ_IND *ind);
```

Parameters:

`ind` This pointer to the indication control structure of the type **MMSREQ_IND** is received in the **u_fopen_ind** function.

Return Value: `ST_RET`

SD_SUCCESS. No Error.

`<> 0` Error Code.

mv_file_open_resp

Usage: This virtual machine function allows the user to respond to a FileOpen indication (**u_fopen_ind**) without having to create an input structure of type **FOPEN_RESP_INFO** ahead of time. The requested file is opened, state information is initialized for the ensuing file transfer, and a FileOpen response is sent back to the requesting MMS-user. An index into the **m_local_fctrl_info** data structure for the open file is returned as an output parameter. A reference to the file control data structure allows the application to monitor the progress of the file transfer outside of the indication function. Note that if this virtual machine function is used to respond to a FileOpen indication, the other virtual machine functions, **mv_fread_resp** and **mv_fclose_resp** should also be used, rather than the paired primitive functions, **mp_fread_resp** and **mp_fclose_resp**.

Function Prototype:

```
ST_RET mv_file_open_resp (MMSREQ_IND *req_info,
                          ST_INT *fctrl_index_out);
```

Parameters:

`req_info` This pointer to the indication control structure of the type **MMSREQ_IND** is received in the **u_fopen_ind** function.

`fctrl_index_out` This is the address of a **ST_INT** that contains the index to the local file control information associated with the open file.

Return Value: `ST_RET`

SD_SUCCESS. No Error.

`<> 0` Error Code.

u_mv_fopen_conf

Usage: This virtual machine user confirmation function is called when a confirm to a virtual machine FileOpen request (**mv_fopen**) is received by a server node. **resp_err** contains a value indicating whether an error occurred. **resp_info_ptr** may contain data that the remote node sent back in the confirmation or it may contain error information if an error occurred. If no error has occurred (**req->resp_err = CNF_RESP_OK**), FileOpen information is available to the application using the **req->resp_info_ptr** pointer.

See **Volume 1 — Module 1 — Virtual User Confirmation Function Class** for more information on the recommended handling of confirms.

Function Prototype: `ST_VOID u_mv_fopen_conf (MMSREQ_PEND *req);`

Parameters:

`req` This pointer to the **MMSREQ_PEND** structure is returned from the original **mv_fopen** request function. See **Volume 1 — Module 2 — Request and Indication Control Data Structures** for more information on this structure.

For a positive response (**req->resp_err = CNF_RESP_OK**), the request control structure member **req->resp_info_ptr** is a pointer to the operation specific data structure (**FOPEN_RESP_INFO**) for the FileOpen function. See page 3-278 for a detailed description of this structure.

For a negative response, or other errors, see **Module 11 — MMS-EASE Error Handling** starting on page 3-339 for more information.

Return Value: `ST_VOID` (ignored)

NOTES:

1. Before returning from any virtual user confirmation function, **ms_clr_mvreq** should be called. This clears up, and frees the data used by the virtual machine to handle the request and confirmation.
2. The contents of this function are completely user-defined.
3. See **mv_fread** on page 3-293 for more information regarding situations that can occur when reading only partial file segments using the virtual machine.

7. FileRead Service

This service is used to transfer all or part of the contents of an open file from a server to a client. It transfers data sequentially from the file position maintained by the **F**ile **R**ead **S**tate **M**achine (**FRSM**), and going to the end of the file.

Paired Primitive Level FileRead Operations

The following section contains information on how to use the paired primitive interface for the FileRead service. It covers available data structures used by the PPI, and the four primitive level functions that together make up the FileRead service. See **Volume 1 — Module 2 — General Sequence of Events — Confirmed Services** for a detailed explanation of the sequence of events that occur within the MMS-EASE environment during confirmed services.

- The FileRead service consists of the paired primitive functions of **`mp_fread`**, **`u_fread_ind`**, **`mp_fread_resp`**, and **`u_mp_fread_conf`**.

Data Structures

Request/Indication

The operation specific data structure described below is used by the client in issuing the FileRead request (**`mp_fread`**). It is received by the server when a FileRead indication (**`u_fread_ind`**) is received.

```
struct fread_req_info
  {
  ST_INT32   frsmid;
  };
typedef struct fread_req_info FREAD_REQ_INFO;
```

Fields:

`frsmid` This contains the **F**ile **R**ead **S**tate **M**achine **ID** (**FRSMID**) of the file to be read. The FRSMID is obtained when the file is opened. See **`mp_fopen_resp`** for more information.

Response/Confirm

The operation specific data structure described below is used by the server in issuing a FileRead response (**`mp_fread_resp`**). It is received by the client when a FileRead confirm (**`u_mp_fread_conf`**) is received.

```
struct fread_resp_info
  {
  ST_INT      fd_len;
  ST_UCHAR   *filedata;
  ST_BOOLEAN more_follows;
  SD_END_STRUCT
  };
typedef struct fread_resp_info FREAD_RESP_INFO;
```

Fields:

`fd_len` This contains the length of file data, in bytes, pointed to by filedata.

`filedata` This is a pointer to the file data to be read.

`more_follows`
- **`SD_TRUE`**. Not the end of the file. More FileRead requests are necessary to complete the file transfer. This is the default.
- **`SD_FALSE`**. End-Of-File. No more data available.

Paired Primitive Interface Functions

mp_fread

Usage: This primitive request function sends a FileRead request PDU using the data from a structure of type **FREAD_REQ_INFO**, pointed to by **info**. The FileRead service is used to read data from a file at the remote node previously opened using **mp_fopen**.

Function Prototype:

```
MMSREQ_PEND *mp_fread (ST_INT chan,
                       FREAD_REQ_INFO *info);
```

Parameters:

chan	This integer contains the channel number over which the FileRead PDU is to be sent.
info	This pointer to a structure of type **FREAD_REQ_INFO** contains information specific to the FileRead PDU to be sent.

Return Value:

MMSREQ_PEND *	This pointer to the request control data structure of type **MMSREQ_PEND** is used to send the FileRead PDU. In case of an error, the pointer is set to null and **mms_op_err** is written with the error code.

Corresponding User Confirmation Function: **u_mp_fread_conf**

Operation-Specific Data Structure Used: **FREAD_REQ_INFO**

See page 3-287 for a detailed description of this structure.

u_fread_ind

Usage: This user indication function is called when a FileRead indication is received by a server node. The contents and operation of this function are user-defined, and depend on the application requirements. To conform to the MMS protocol requirements, the application must do one of the following some time after the indication is received:

1) send a positive response using either:

   a) the primitive response function (**`mp_fread_resp`**) after the file data has been successfully read, or

   b) the virtual machine response function (**`mv_fread_resp`**) to have the virtual machine take the requested action and send the response, or

2) call the appropriate primitive negative (error) response function (**`mp_err_resp`**). See **Module 11 — MMS-EASE Error Handling** on page 3-343 for more information on this function.

See **Volume 1 — Module 1 — User Indication Function Class** for additional information on the recommended handling of indications.

Function Prototype: `ST_VOID u_fread_ind  (MMSREQ_IND *ind);`

Parameters:

`ind` This pointer to the **`MMSREQ_IND`** structure is described in **Volume 1 — Module 2 — Request and Indication Control Data Structures**. This pointer must be used to send the response to this indication. The primary item of interest in this structure is **`ind->resp_info_ptr`**. This is a pointer to the operation specific data structure for a File Read indication (**`FREAD_REQ_INFO`**). This pointer will always be valid when **`u_fread_ind`** is called.

Return Value: `ST_VOID` (ignored)

Operation-Specific Data Structure Used: **`FREAD_REQ_INFO`**

See page 3-287 for a detailed description of this structure.

mp_fread_resp

Usage: This primitive response function sends a FileRead positive response PDU using the data from a structure of type **FREAD_RESP_INFO**, pointed to by **info**. This function should be called after the **u_fread_ind** function is called (a FileRead indication was received), and after the file data has been successfully read.

Function Prototype:

```
ST_RET mp_fread_resp (MMSREQ_IND *ind,
                      FREAD_RESP_INFO *info);
```

Parameters:

`ind` This pointer to the indication control structure of type **MMSREQ_IND** is received in the **u_fread_ind** function.

`info` This pointer to an Operation Specific data structure of type **FREAD_RESP_INFO** contains information specific to the FileRead response PDU to be sent.

Return Value: `ST_RET`

SD_SUCCESS. No Error.

`<> 0` Error Code.

Corresponding User Indication Function: **u_fread_ind**

Operation-Specific Data Structure Used: **FREAD_RESP_INFO**

See page 3-287 for a detailed description of this structure.

u_mp_fread_conf

Usage: This primitive user confirmation function is called anytime a confirmation to a primitive File Read request (**mp_fread**) is received. **resp_err** contains a value indicating whether an error occurred. **resp_info_ptr** may contain data that the remote node sent back in the confirmation or it may contain error information if an error occurred. If no error has occurred (**req->resp_err = CNF_RESP_OK**), FileRead information is available to the application using the **req->resp_info_ptr** pointer. See **Volume 1 — Module 1 — Primitive User Confirmation Function Class** for more information on the recommended handling of confirms.

Note that the function called for this event may be changed if desired; however, **u_mp_fread_conf** is the standard default function.

Function Prototype:

```
ST_VOID u_mp_fread_conf (MMSREQ_PEND *req);
```

Parameters:

`req` This pointer to the **MMSREQ_PEND** structure is returned from the original **mp_fread** request function. See **Volume 1 — Module 2 — Request and Indication Control Data Structures** for more information on this structure.

For a positive response (**req->resp_err = CNF_RESP_OK**), the request control structure member **resp_info_ptr** is a pointer to the operation specific data structure (**FREAD_RESP_INFO**) for the FileRead function.

For a negative response, or other errors, see **Module 11 - MMS-EASE Error Handling** starting on page 3-323 for more information.

Return Value: `ST_VOID` (ignored)

Operation-Specific Data Structure Used: **FREAD_RESP_INFO**

See page 3-287 for a detailed description of this structure.

Virtual Machine FileRead Operations

The following section information on how to use the virtual machine interface for the FileRead service. It covers available data structures used by the VMI, and the virtual machine functions that together make up the File Read service.

- The FileRead service consists of the virtual machine functions of **`mv_fread`**, **`mv_fread_resp`**, and **`u_mv_fread_conf`**.

In the virtual machine confirm function, **`u_mv_fread_conf`**, it uses the **`FREAD_RESP_INFO`** data structure.

Virtual Machine Interface Functions

mv_fread

Usage: This virtual machine function allows the user to execute the FileRead service without having to create an input structure of type **FREAD_REQ_INFO** ahead of time. This function may make multiple calls to the **mp_fread** primitive function. This is so that the entire file (or as much of it as desired) is transferred before the user-defined function (**u_mv_fread_conf**) is called. Within the **u_mv_fread_conf** function, the file data may be written to a file. Another option is that **mv_fread** may be called again with different values for **dest** and **num_bytes** if there is more data in the file.

Function Prototype:

```
MMSREQ_PEND *mv_fread  (ST_INT chan,
                        ST_INT refnum,
                        ST_UCHAR *dest,
                        ST_LONG num_bytes);
```

Parameters:

chan	This indicates the channel on which file transfer is taking place.
refnum	This reference number is acquired from the same **FCTRL_INFO** structure as the **num_bytes** value.
dest	This is a pointer to the buffer in memory where the file segment is to be stored.
num_bytes	This indicates the number of bytes to transfer. This must be equal to the number of bytes in the **dest** buffer.

Return Value:

MMSREQ_PEND *	This pointer to a request control data structure of type **MMSREQ_PEND** holds information regarding this request. In case of an error, the pointer is set to null and **mms_op_err** is written with the error code.

Corresponding Confirmation Function: **u_mv_fread_conf**

NOTE: If the virtual machine is used to read less than one file segment or more than one but with only a fractional file segment read *during the last FileRead*, some of the file data will not be written into the destination pointer. This occurs because the remote node sends back more data than was requested of the virtual machine. The virtual machine assumes the destination buffer pointed to by **dest** is only as big as the number of bytes to read. Therefore, the virtual machine has no place to put the residual data that cannot fit into the destination buffer. On a subsequent FileRead, the remote node sends back data from where it left off. This may result in some portion of the file not being written into the destination buffer. To avoid this problem, the data, pointed to by **resp_info_ptr**, can be examined. This is a structure of type **FREAD_RESP_INFO** (within **u_mv_fread_conf**). It can be used to access data that may not have been stored at **dest**. Also, the local node has little control over the size of the file segments that the remote node sends back. It is strongly suggested that, if only partial file segments are read, you subsequently close the file, then reopen it using a different initial position. Another suggestion is to use the PPI instead.

mv_fread_resp

Usage: This virtual machine function allows the user to respond to a FileRead indication (**`u_fread_ind`**) without having to create an input structure of type **`FREAD_RESP_INFO`** explicitly. This function automatically retrieves the requested file segment, and sends out a FileRead response. It also takes care of managing the necessary state information regarding this file transfer.

Function Prototype: `ST_RET mv_fread_resp (MMSREQ_IND *ind);`

Parameters:

`ind` This pointer to a indication control data structure of the type **`MMSREQ_IND`** is received for the File Read indication.

Return Value: `ST_RET` **`SD_SUCCESS`**. No Error.

`<> 0` Error Code.

u_mv_fread_conf

Usage: This virtual user confirmation function is called when a confirm to a virtual FileRead request (**mv_fread**) is received. **resp_err** contains a value indicating whether an error occurred. **resp_info_ptr** may contain data that the remote node sent back in the confirmation or it may contain error information if an error occurred. If no error has occurred (**req->resp_err = CNF_RESP_OK**), FileRead information is available to the application using the **req->resp_info_ptr** pointer. See **Volume 1 — Module 1 — Virtual User Confirmation Function Class** for more information on the recommended handling of confirms.

Note that the function called for this event may be changed if desired; however, **u_mv_fread_conf** is the standard default function.

Function Prototype:

```
ST_VOID u_mv_fread_conf (MMSREQ_PEND *req);
```

Parameters:

`req` This pointer to the **MMSREQ_PEND** structure is returned from the original **mv_fread** request function. See **Volume 1 — Module 2 — Request and Indication Control Data Structures** for more information on this structure.

For a positive response (**req->resp_err = CNF_RESP_OK**), the request control structure member **resp_info_ptr** is a pointer to the operation specific data structure **FREAD_RESP_INFO** for the FileRead function. See page 3-287 for a detailed description of this structure.

For a negative response, or other errors, see **Module 11— MMS-EASE Error Handling** starting on page 3-339 for more information.

Return Value: `ST_VOID` (ignored)

NOTES:

1. Before returning from any virtual user confirmation function, **ms_clr_mvreq** should be called. This clears up and frees the data used by the virtual machine to handle the request and confirmation.
2. The contents of this function are completely user-defined.
3. See **mv_fread** on page 3-293 for more information regarding situations that can occur when reading only partial file segments using the virtual machine.

8. FileClose Service

This service is used to request that a specified file be closed, and all resources associated with the file transfer be released. A successful FileClose causes the corresponding **F**ile **R**ead **S**tate **M**achine (**FRSM**) to be deleted, and the FRSMID is available for reassignment.

Paired Primitive Level FileClose Operations

The following section contains information on how to use the paired primitive interface for the FileClose service. It covers available data structures used by the PPI, and the four primitive level functions that together make up the FileClose service. See **Volume 1 — Module 2 — General Sequence of Events — Confirmed Services** for a detailed explanation of the sequence of events that occur within the MMS-EASE environment during confirmed services.

- The FileClose service consists of the paired primitive functions of **`mp_fclose`**, **`u_fclose_ind`**, **`mp_fclose_resp`**, and **`u_mp_fclose_conf`**.

Data Structures

Request/Indication

The operation specific data structure described below is used by the client in issuing the FileClose request (**`mp_fclose`**). It is received by the server when a FileClose indication (**`u_fclose_ind`**) is received.

```
struct fclose_req_info
  {
  ST_INT32   frsmid;
  };
typedef struct fclose_req_info FCLOSE_REQ_INFO;
```

Fields:

`frsmid` This contains the **F**ile **R**ead **S**tate **M**achine **ID** (FRSMID) obtained when the file was opened using a call to **`mp_fopen`**.

Paired Primitive Interface Functions

mp_fclose

Usage: This primitive request function sends a FileClose request PDU using the File Read State Machine ID (FRSMID). This is a member of the **FCLOSE_REQ_INFO** structure pointed to by **info**. This service is used to close a file previously opened for reading.

Function Prototype:
```
MMSREQ_PEND *mp_fclose (ST_INT chan,
                        FCLOSE_REQ_INFO *info);
```

Parameters:

chan	This integer contains the channel number over which the FileClose PDU is to be sent.
info	This pointer to the operation specific data structure of type **FCLOSE_REQ_INFO** contains the information specific to the FileClose PDU.

Return Value:

MMSREQ_PEND *	This pointer to the request control data structure of type **MMSREQ_PEND** is used to send the FileClose PDU. In case of an error, the pointer is set to null and **mms_op_err** is written with the error code.

Corresponding User Confirmation Function: **u_mp_fclose_conf**

Operation-Specific Data Structure Used: **FCLOSE_REQ_INFO**

See page 3-297 for a detailed description of this structure.

NOTE: Do not close a file on the remote node using a call to **mp_fclose** if the file was originally opened using the virtual machine call to **mv_fopen**. Failure to follow this restriction will cause erroneous virtual machine file operations.

u_fclose_ind

Usage: This user indication function is called when a FileClose indication is received by a server node. The contents and operation of this function are user-defined, and depend on the application requirements. To conform to the MMS protocol requirements, the application must do one of the following some time after the indication is received:

1) send a positive response using either:

   a) the primitive response function (**`mp_fclose_resp`**) after the file is closed successfully, or

   b) the virtual machine response function (**`mv_fclose_resp`**) to have the virtual machine take the requested action and send the response, or

2) call the appropriate primitive negative (error) response function (**`mp_err_resp`**). See **Module 11 — MMS-EASE Error Handling** on page 3-343 for more information on this function.

See **Volume 1 — Module 1 — User Indication Function Class** for additional information regarding the recommended handling of indications.

Function Prototype: `ST_VOID u_fclose_ind (MMSREQ_IND *ind);`

Parameters:

`ind` This pointer to the **`MMSREQ_IND`** structure is described in **Volume 1 — Module 2 — Request and Indication Control Data Structures**. This pointer must be used to send the response to this indication. The primary item of interest in this structure is **`ind->resp_info_ptr`**. This is a pointer to the operation specific data structure for a FileRead indication (**`FCLOSE_REQ_INFO`**). This pointer will always be valid when **`u_fclose_ind`** is called.

Return Value: `ST_VOID` (ignored)

Operation-Specific Data Structure Used: **`FCLOSE_REQ_INFO`**

See page 3-297 for a detailed description of this structure.

mp_fclose_resp

Usage: This primitive response function sends a FileClose positive response PDU. This function should be called after the **`u_fclose_ind`** function is called (a FileClose indication is received), and after the file is actually closed successfully.

Function Prototype: `ST_RET mp_fclose_resp (MMSREQ_IND *ind);`

Parameters:

`ind` This pointer to the indication control data structure of type **`MMSREQ_IND`** is passed to the **`u_fclose_ind`** function when it was called.

Return Value: `ST_RET` **`SD_SUCCESS`**. No Error.

`<> 0` Error Code.

Corresponding User Indication Function: **`u_fclose_ind`**

Operation-Specific Data Structure Used: NONE

u_mp_fclose_conf

Usage: This primitive user confirmation function is called anytime a confirmation to a primitive FileClose request (**mp_fclose**) is received. **resp_err** contains a value indicating whether an error occurred. **resp_info_ptr** may contain data that the remote node sent back in the confirmation or it may contain error information if an error occurred. See **Volume 1 — Module 1 — Primitive User Confirmation Function Class** for more information on the recommended handling of confirms.

Note that the function called for this event may be changed, if desired; however, **u_mp_fclose_conf** is the standard default function.

Function Prototype:

```
ST_VOID u_mp_fclose_conf (MMSREQ_PEND *req);
```

Parameters:

`req` This pointer to the **MMSREQ_PEND** structure is returned from the original **mp_fclose** request function. See **Volume 1 — Module 2 — Request and Indication Control Data Structures** - for more information on this structure.

For a negative response, or other errors, see **Module 11 — MMS-EASE Error Handling** starting on page 3-339 for more information.

Return Value: `ST_VOID` (ignored)

Operation Specific Data Structure Used: NONE

Virtual Machine FileClose Operations

The following section information on how to use the virtual machine interface for the FileClose service. It covers the virtual machine functions that together make up the VMI FileClose service.

- The FileClose service consists of the virtual machine functions of **mv_fclose**, **mv_fclose_resp**, and **u_mv_fclose_conf**.

There are no virtual machine data structures.

Virtual Machine Interface Functions

mv_fclose

Usage: This virtual machine function allows the user to execute the FileClose service (close a remote file) without explicitly having to create an input structure of type **FCLOSE_REQ_INFO**. The necessary state information regarding this file is checked before sending the FileClose request and is updated when the confirmation is received.

Function Prototype: `MMSREQ_PEND *mv_fclose (ST_INT chan, ST_INT refnum);`

Parameters:

`chan` This indicates the channel on which the file was opened.

`refnum` This reference number is acquired from the same **FCTRL_INFO** structure as the **bytes_read** value.

Return Value:

`MMSREQ_PEND *` This pointer to a request control data structure of type **MMSREQ_PEND** holds information regarding this request. In case of an error, the pointer is set to null and **mms_op_err** is written with the error code.

Corresponding Confirmation Function: **u_mv_fclose_conf**

Data Structures Used: **m_rem_fctrl_info**

See page 3-260 for a detailed description of this structure.

mv_fclose_resp

Usage: This virtual machine function allows the user to respond to a FileClose indication (**u_fclose_ind**) without having to create explicitly an input structure of type **FCLOSE_RESP_INFO**. The file being read is closed, the structure containing the state information regarding this file transfer is written, and the FileClose response is sent.

Function Prototype: `ST_RET mv_fclose_resp (MMSREQ_IND *ind);`

Parameters:

`ind` This pointer to a indication control data structure of the type **MMSREQ_IND** is received for the FileClose indication.

Return Value: `ST_RET` **SD_SUCCESS**. No Error.

`<> 0` Error Code.

NOTE: An error response is sent back to the requester if the file requested to be closed was not open.

u_mv_fclose_conf

Usage: This virtual user confirmation function is called when a confirm to a virtual machine File-Close request (**mv_fclose**) is received. **resp_err** contains a value indicating whether an error occurred. **resp_info_ptr** may contain data that the remote node sent back in the confirmation or it may contain error information if an error occurred. See **Volume 1 — Module 1 — Virtual User Confirmation Function Class** for more information on the recommended handling of confirms.

Note that the function called for this event may be changed if desired; however, **u_mv_fclose_conf** is the standard default function.

Function Prototype: `ST_RET mv_fclose_resp (MMSREQ_IND *ind);`

Parameters:

`req` This pointer to the **MMSREQ_PEND** structure is returned from the original **mv_fclose** request function. See **Volume 1 — Module 2 — Request and Indication Control Data Structures** for more information on this structure.

For a negative response, or other errors, see **Module 11 — MMS-EASE Error Handling** starting on page 3-339 for more information.

Return Value: `ST_VOID` (ignored)

Operation Specific Data Structure Used: NONE

NOTES:

1. Before returning from any virtual user confirmation function, **ms_clr_mvreq** should be called. This clears up and frees the data used by the virtual machine to handle the request and confirmation.
2. The contents of this function are completely user-defined.

9. FileCopy Service

FileCopy is not a true MMS service. Rather it is a MMS-EASE service that automatically generates MMS FileOpen, FileRead, and FileClose PDUs. The FileCopy service allows a client to request that a specified file be copied from the virtual filestore of a server to the virtual filestore of the client. It will overwrite any existing file with the same name that is already present in the client's filestore.

Paired Primitive Level FileCopy Operations

There is no primitive level interface to be used with the FileCopy service. The Virtual Machine Interface must be used. See below.

Virtual Machine FileCopy Operations

The following section contains information on how to use the virtual machine interface for the FileCopy service. It covers available data structures used by the VMI, and virtual machine functions that together make up this service.

- The FileCopy service consists of the virtual machine functions of **`mv_fcopy`** and **`u_mv_fcopy_conf`**. This internally handles the transmittal of a FileOpen, FileRead, and FileClose.

Data Structures

This structure allows the user application program to examine information managed by the virtual machine during file copy operations (**`mv_fcopy`**). **THIS INFORMATION IS INTENDED TO BE VISIBLE IN A READ-ONLY CAPACITY. MODIFYING THIS STRUCTURE CAN PRODUCE UNPREDICTABLE RESULTS.**

Its use is similar to that of an operation-specific data structure. The **`mv_fcopy`** function returns a pointer to a **`MMSREQ_PEND`** structure. If this pointer is non-null, the **`req_info_ptr`** member of the **`MMSREQ_PEND`** structure will point to a **`FCOPY_INFO`** structure.

```
struct fcopy_info
  {
  ST_INT           opcode;
  ST_RET           error;
  OBTFILE_REQ_INFO *obtfinfo;
  FCTRL_INFO       *fctrl;
  struct fcopy_pipe_ctrl *fpipe;
  };
typedef struct fcopy_info FCOPY_INFO;
```

Fields:

`opcode` This contains the MMS operation code, **`MMSOP_REM_FILE_COPY`**. This operation code is documented in the **mmsop_en.h** file.

`error` This error code is used by the virtual machine. **FOR INTERNAL USE ONLY — DO NOT USE**.

`obtfinfo` This is a pointer to a structure that is the same type as the operation specific data structure used in the ObtainFile service, **`OBTFILE_REQ_INFO`**. See page 3-269 for a detailed description of this structure.

`fctrl` This member is a pointer to the file control structure, **`FCTRL_INFO`**. It points to the same information that can be referenced from the **`rem_fctrl_info.finfo`** array structure. In particular, it allows accessing the **`bytes_read`** value during a file copy operation. However, this information is only valid if the **`stat`** member of the **`rem_fctrl_info`** structure is NOT equal to **`NO_ACT`**. This is because the **`fctrl_info`** structure is stored in dynamic memory. It will be freed by the virtual machine during a call to **`ms_comm_serve`** if the virtual machine needs the space. The following could be used to test the state flag before accessing the **`bytes_read`** value:

```
if (rem_fctrl_info.stat[refnum] != NO_ACT)
   {
   .
   .
   };
```

where:

`refnum` is acquired from the same **`FCTRL_INFO`** structure as the **`num_bytes`** value. This value needs to be saved after the initial call to **`mv_fcopy`**, and before the first call to **`ms_comm_serve`**. This is because it is also stored in dynamic memory. Valid states for the **`rem_fctrl_info.stat`** array are:

```
#define MFATAT_NO_ACT      0      No activity
#define MFSTAT_PEND_OPEN   1      Pending Open
#define MFSTAT_OPENED      2      Opened
#define MFSTAT_PEND_CLOSE  3      Pending Close
```

`fpipe` This member is a pointer to the file pipeline structure, **`fcopy_pipe_ctrl`**. See page 3-260 for more information.

Virtual Machine Interface Functions

mv_fcopy

Usage: This virtual machine function allows the user to copy a file from a remote node's file system to the local file system. This can be done without having to generate and manage the individual requests, confirmations, responses, required by the MMS file operations or the operating system calls necessary to create the file locally. The necessary state information (in **m_rem_fctrl_info** and **m_local_fctrl_info**) regarding this file transfer is initialized, and maintained by the virtual machine automatically.

Function Prototype:

```
MMSREQ_PEND *mv_fcopy  (ST_INT chan,
                        ST_CHAR *from,
                        ST_INT from_len,
                        ST_CHAR *to,
                        ST_INT to_len);
```

Parameters:

chan	This indicates the channel on which the file transfer is to take place.
from	This is a pointer to the path and file name of the remote file to be copied to the local user's file system.
from_len	This indicates the length, in bytes, of the source file and path name.
to	This is a pointer to the path and name of the file to be created locally.
to_len	This indicates the length, in bytes, of the destination file and path name.

Return Value:

MMSREQ_PEND *	This pointer to a request control data structure of type **MMSREQ_PEND** holds information regarding this request. If this points to a non-null value, then the virtual machine is attempting to copy a file. The information being sent can be examined using the **FCOPY_INFO** structure. See page 3-305 for more information. In case of an error, the pointer is set to null and **mms_op_err** is written with the error code.

Corresponding Confirmation Function: **u_mv_fcopy_conf**

NOTES:

1. The following sequence of actions takes place when this function is executed:
   a. The remote and local files are opened
   b. The remote file is read into the local file
   c. The remote and local files are closed
2. If the local file specified exists, it is overwritten with the contents of the remote file as it is being read.
3. Pipeline depth can be specified for a FileCopy request. See page 3-260 for more information.

u_mv_fcopy_conf

Usage: This virtual user confirmation function is called when the confirm to a virtual FileCopy request (**mv_fcopy**) completes. **resp_err** contains a value indicating whether an error occurred. **resp_info_ptr** may contain data regarding this FileCopy, or it may contain error information if an error occurred.

Function Prototype: `ST_VOID u_mv_fcopy_conf (MMSREQ_PEND *req);`

Parameters:

`req` This pointer to the **MMSREQ_PEND** is returned from the original **mv_fcopy** request function. See **Volume 1 — Module 2 — Request and Indication Control Data Structures** for more information on this structure.

For a negative response, or other errors, see **Module 11 — MMS-EASE Error Handling** starting on page 3-339 for more information.

Return Value: `ST_VOID` (ignored)

Operation Specific Data Structure Used: **FCTRL_INFO**

See page 3-258 for a detailed description of this structure.

NOTES:

1. Before returning from any virtual user confirmation function, **ms_clr_mvreq** should be called. This clears up and frees the data used by the virtual machine to handle the request and confirmation.
2. The contents of this function are completely user-defined.
3. FileCopy is not a MMS service. It is a virtual machine service that uses the MMS FileOpen, FileRead, and FileClose services.

10. FileRename Service

This service is used by a client to request that a specified file's name be changed in the virtual filestore of a server.

Paired Primitive Level FileRename Operations

The following section contains information on how to use the paired primitive interface for the FileRename service. It covers available data structures used by the PPI, and the four primitive level functions that together make up the FileRename service. See **Volume 1 — Module 2 — General Sequence of Events — Confirmed Services** for a detailed explanation of the sequence of events that occur within the MMS-EASE environment during confirmed services.

- The FileRename service consists of the paired primitive functions of **`mp_frename`**, **`u_frename_ind`**, **`mp_frename_resp`**, and **`u_mp_frename_conf`**.

Data Structures

Request/Indication

The operation specific data structure described below is used by the client in issuing the FileRename request (**`mp_frename`**). It is received by the server when a FileRename indication (**`u_frename_ind`**) is received.

```
struct frename_req_info
  {
  ST_INT    num_of_cur_fname;
  ST_INT    num_of_new_fname;
/*FILE_NAME cur_fname_list[ num_of_cur_fname ];                    */
/*FILE_NAME new_fname_list[ num_of_new_fname ];                    */
  SD_END_STRUCT
  };
typedef struct frename_req_info  FRENAME_REQ_INFO;
```

Fields:

`num_of_cur_fname`	This contains the number of elements in the file name sequence specifying the current name of the file to be renamed.
`num_of_new_fname`	This contains the number of elements in the file name sequence specifying the new name of the file.
`cur_fname_list`	This list of structures of type **`FILE_NAME`** contains the elements that comprise the file name sequence. This specifies the current name of the file to be renamed.
`new_fname_list`	This list of structures of type **`FILE_NAME`** contains the elements that comprise the file name sequence. It specifies the new name of the file.

NOTES:

1. See the description of the file name structure on page 3-267 for more information on the format of MMS file names.
2. FOR REQUESTS ONLY, when allocating a data structure of type **`FRENAME_REQ_INFO`**, enough memory must be allocated to hold the information for the file name sequences. The following C statement can be used:

```
info = (FRENAME_REQ_INFO *) chk_malloc(sizeof(FRENAME_REQ_INFO) +
             (num_of_cur_fname + num_of_new_fname) * sizeof(FILE_NAME));
```

Paired Primitive Interface Functions

mp_frename

Usage: This primitive request function sends a FileRename request PDU using the data from a structure of type **FRENAME_REQ_INFO**, pointed to by **info**. The FileRename service is used to rename a file at the remote node. The file to rename should not be open for reading.

Function Prototype:

```
MMSREQ_PEND *mp_frename (ST_INT chan,
                         FRENAME_REQ_INFO *info);
```

Parameters:

chan	This integer contains the channel number over which the FileRename PDU is to be sent.
info	This pointer to a structure of type **FRENAME_REQ_INFO** contains information specific to the FileRename PDU to be sent.

Return Value:

MMSREQ_PEND *	This pointer to the request control data structure of type **MMSREQ_PEND** is used to send the FileRename PDU. In case of an error, the pointer is set to null and **mms_op_err** is written with the error code.

Corresponding User Confirmation Function: **u_mp_frename_conf**

Operation-Specific Data Structure Used: **FRENAME_REQ_INFO**

See page 3-309 for a detailed description of this structure.

u_frename_ind

Usage: This user indication function is called when a FileRename indication is received by a server node. The contents and operation of this function are user-defined, and depend on the application requirements. To conform to the MMS protocol, the application must do one of the following some time after the indication is received:

1) send a positive response using either:

   a) the primitive response function (**`mp_frename_resp`**) after successfully renaming the file, or

   b) the virtual machine response function (**`mv_frename_resp`**) to have the virtual machine take the requested action and send the response, or

2) call the appropriate primitive negative (error) response function (**`mp_err_resp`**). See **Module 11 — MMS-EASE Error Handling** starting on page 3-343 for more information on this function.

See **Volume 1 — Module 1 — User Indication Function Class** for additional information regarding the recommended handling of indications.

Function Prototype: `ST_VOID u_frename_ind (MMSREQ_IND *ind);`

Parameters:

`ind` This pointer to the **`MMSREQ_IND`** structure is described in **Volume 1 — Module 2 — Request and Indication Control Data Structures**. This pointer must be used to send the response to this indication. The primary item of interest in this structure is **`ind->resp_info_ptr`**. This is a pointer to the operation specific data structure for a File Rename indication (**`FRENAME_REQ_INFO`**). This pointer will always be valid when **`u_frename_ind`** is called.

Return Value: `ST_VOID` (ignored)

Operation-Specific Data Structure Used: **`FRENAME_REQ_INFO`**

See page 3-309 for a detailed description of this structure.

mp_frename_resp

Usage: This primitive response function sends a FileRename positive response PDU. This function should be called as a response to the **u_frename_ind** function being called (a FileRename indication is received), and after successfully renaming the file.

Function Prototype: `ST_RET mp_frename_resp (MMSREQ_IND *ind);`

Parameters:

`ind` This pointer to the indication control data structure of type **MMSREQ_IND** is passed to the **u_frename_ind** function when it was called.

Return Value: `ST_RET` **SD_SUCCESS**. No Error.

`<> 0` Error Code.

Corresponding User Indication Function: **u_frename_ind**

Operation-Specific Data Structure Used: NONE

u_mp_frename_conf

Usage: This primitive user confirmation function is called when a confirmation to a primitive FileRename request (**mp_frename**) is received. **resp_err** contains a value indicating whether an error occurred. **resp_info_ptr** may contain data that the remote node sent back in the confirmation or it may contain error information if an error occurred. See **Volume 1 — Module 1 — Primitive User Confirmation Function Class** for more information on the recommended handling of confirms.

Note that the function called for this event may be changed, if desired; however, **u_mp_frename_conf** is the standard default function.

Function Prototype:

```
ST_VOID u_mp_frename_conf (MMSREQ_PEND *req);
```

Parameters:

`req` This pointer to the **MMSREQ_PEND** structure is returned from the original **mp_frename** request function. See **Volume 1 — Module 2 — Request and Indication Control Data Structures** for more information on this structure.

For a negative response, or other errors, see **Module 11 — MMS-EASE Error Handling** starting on page 3-339 for more information.

Return Value: `ST_VOID` (ignored)

Operation Specific Data Structure Used: NONE

Virtual Machine FileRename Operations

The following section information on how to use the virtual machine interface for the FileRename service.

- The FileRename service consists of the virtual machine function of **`mv_frename_resp`**.

There are no virtual machine data structures.

Virtual Machine Interface Functions

mv_frename_resp

Usage: This virtual machine function allows the user to respond to a FileRename indication (**`u_frename_ind`**). This occurs without having to interact with the operating system directly to rename the file.

Function Prototype: `ST_RET mv_frename_resp (MMSREQ_IND *ind);`

Parameters:

`ind` This pointer to a indication control data structure of the type **`MMSREQ_IND`** is received for the File Rename indication.

Return Value: `ST_RET` **`SD_SUCCESS`**. No Error.

`<> 0` Error Code.

NOTE: An error response is sent back to the requester if the requested file to be renamed was open.

11. FileDelete Service

This service is used by a client to delete a file from the virtual filestore of a server.

Paired Primitive Level FileDelete Operations

The following section contains information on how to use the paired primitive interface for the FileDelete service. It covers available data structures used by the PPI, and the four primitive level functions that together make up the FileDelete service. See **Volume 1 — Module 2 — General Sequence of Events — Confirmed Services** for a detailed explanation of the sequence of events that occur within the MMS-EASE environment during confirmed services.

- The FileDelete service consists of the paired primitive functions of **`mp_fdelete`**, **`u_fdelete_ind`**, **`mp_fdelete_resp`**, and **`u_mp_fdelete_conf`**.

Data Structures

Request/Indication

The operation specific data structure described below is used by the client in issuing the FileDelete request (**`mp_fdelete`**). It is received by the server when a FileDelete indication (**`u_fdelete_ind`**) is received.

```
struct fdelete_req_info
  {
  ST_INT    num_of_fname;
/*FILE_NAME fname_list[ num_of_fname ];             */
  SD_END_STRUCT
  };
typedef struct fdelete_req_info  FDELETE_REQ_INFO;
```

Fields:

`num_of_fname`	This indicates the number of elements in the file name sequence specifying the name of the file to be deleted.
`fname_list`	This list of structures of type **`FILE_NAME`** contains the elements that comprise the file name sequence. This indicates the name of the file to be deleted.

NOTES:

1. See the description of the file name structure on page 3-267 for more information on the format of MMS file names.
2. When allocating a data structure of type **`FDELETE_REQ_INFO`**, enough memory must be allocated to hold the information for the file name sequence. The following C statement can be used:

```
info = (FDELETE_REQ_INFO *) chk_malloc(sizeof(FDELETE_REQ_INFO) + num_of_fname *
            (sizeof(FILE_NAME)));
```

Paired Primitive Interface Functions

mp_fdelete

Usage: This primitive request function sends a FileDelete request PDU. The FileDelete service is used to delete a file at the remote node. The file at the remote node should not be open when it is deleted.

Function Prototype:
```
MMSREQ_PEND *mp_fdelete (ST_INT chan,
                         FDELETE_REQ_INFO *info);
```

Parameters:

`chan`	This integer contains the channel number over which the FileDelete PDU is to be sent.
`info`	This pointer to the operation specific data structure of type **`FDELETE_REQ_INFO`** contains the path and file name information of the remote file to be deleted.

Return Value:

`MMSREQ_PEND *`	This pointer to the request control data structure of type **`MMSREQ_PEND`** is used to send the FileDelete PDU. In case of an error, the pointer is set to null and **`mms_op_err`** is written with the error code.

Corresponding User Confirmation Function: **`u_mp_fdelete_conf`**

Operation-Specific Data Structure Used: **`FDELETE_REQ_INFO`**

See page 3-315 for a detailed description of this structure.

u_fdelete_ind

Usage: This user indication function is called when a FileDelete indication is received by a server node. The contents and operation of this function are user-defined, and depend on the application requirements. To conform to the MMS protocol requirements, the application must do one of the following some time after the indication is received:

1) send a positive response using either:

   a) the primitive response function (**`mp_fdelete_resp`**) after successfully deleting a file, or

   b) the virtual response function (**`mv_fdelete_resp`**) to have the virtual machine take the requested action and send the response, or

2) call the appropriate primitive negative (error) response function (**`mp_err_resp`**). See **Module 11 — MMS-EASE Error Handling** on page 3-343 for more information on this structure.

See **Volume 1 — Module 1 — User Indication Function Class** for additional information regarding the recommended handling of indications.

Function Prototype: `ST_VOID u_fdelete_ind (MMSREQ_IND *ind);`

Parameters:

`ind` This pointer to the **`MMSREQ_IND`** structure is described in **Volume 1 — Module 2 — Request and Indication Control Data Structures**. This pointer must be used to send the response to this indication. The primary item of interest in this structure is **`ind->resp_info_ptr`**. This is a pointer to the operation specific data structure for a FileDelete indication (**`FDELETE_REQ_INFO`**). This pointer will always be valid when **`u_fdelete_ind`** is called.

Return Value: `ST_VOID` (ignored)

Operation-Specific Data Structure Used: **`FDELETE_REQ_INFO`**

See page 3-315 for a detailed description of this structure.

mp_fdelete_resp

Usage: This primitive response function sends a FileDelete positive response PDU. This function should be called as a response to the **u_fdelete_ind** function being called (a FileDelete indication is received), and after successfully deleting the file.

Function Prototype: `ST_RET mp_fdelete_resp (MMSREQ_IND *info);`

Parameters:

`info` This pointer to the indication control data structure of type **MMSREQ_IND** is passed to the **u_fdelete_ind** function when it was called.

Return Value: `ST_RET` **SD_SUCCESS**. No Error.

`<> 0` Error Code.

Corresponding User Indication Function: **u_fdelete_ind**

Operation-Specific Data Structure Used: NONE

u_mp_fdelete_conf

Usage: This primitive user confirmation function is called when a confirm to a primitive FileDelete request (**mp_fdelete**) is received. **resp_err** contains a value indicating whether an error occurred. **resp_info_ptr** may contain data that the remote node sent back in the confirmation, or it may contain error information if an error occurred. See **Volume 1 — Module 1 — Primitive User Confirmation Function Class** for more information on the recommended handling of confirms.

Note that the function called for this event may be changed if desired; however, **u_mp_fdelete_conf** is the standard default function.

Function Prototype:

```
ST_VOID u_mp_fdelete_conf (MMSREQ_PEND *req);
```

Parameters:

`req` This pointer to the **MMSREQ_PEND** structure is returned from the original **mp_fdelete** request function. See **Volume 1 — Module 2 — Request and Indication Control Data Structures** for more information on this structure.

For a negative response, or other errors, see **Module 11 — MMS-EASE Error Handling** starting on page 3-339 for more information.

Return Value: `ST_VOID` (ignored)

Operation Specific Data Structure Used: NONE

Virtual Machine FileDelete Operations

The following section information on how to use the virtual machine interface for the FileDelete Service.

- The FileDelete service consists of the virtual machine function of **`mv_fdelete_resp`**.

There are no virtual machine data structures.

Virtual Machine Interface Functions

mv_fdelete_resp

Usage: This virtual machine function allows the user to respond to a FileDelete indication (**`u_fdelete_ind`**) without actually having to perform the file deletion and generate the request using a primitive response function. The requested file is deleted, the file state information is updated and a FileDelete response is sent back to the requesting MMS-user.

Function Prototype: `ST_RET mv_fdelete_resp (MMSREQ_IND *ind);`

Parameters:

`ind` This pointer to a indication control data structure of the type **`MMSREQ_IND`** is received for the FileDelete indication.

Return Value: `ST_RET` **`SD_SUCCESS`**. No Error.

`<> 0` Error Code.

NOTE: An error response is sent back to the requester if the file that was requested to be deleted was open.

12. FileDirectory Service

This service is used by a client to obtain the name and attributes of a file or group of files in the server's file-store. The attributes returned by this service are the same as those returned in the FileOpen service.

Paired Primitive Level FileDirectory Operations

The following section contains information on how to use the paired primitive interface for the FileDirectory service. It covers available data structures used by the PPI, and the four primitive level functions that together make up the FileDirectory service. See **Volume 1 — Module 2 — General Sequence of Events — Confirmed Services** for a detailed explanation of the sequence of events that occur within the MMS-EASE environment during confirmed services.

- The FileDirectory service consists of the paired primitive functions of **mp_fdir**, **u_fdir_ind**, **mp_fdir_resp**, and **u_mp_fdir_conf**.

Data Structures

Request/Indication

The operation specific data structure described below is used by the client in issuing the FileDirectory request (**mp_fdir**). It is received by the server when a FileDirectory indication (**u_fdir_ind**) is received.

```
struct fdir_req_info
  {
  ST_BOOLEAN filespec_pres;
  ST_BOOLEAN cont_after_pres;
  ST_INT     num_of_fs_fname;
  ST_INT     num_of_ca_fname;
/*FILE_NAME  fs_fname_list[ num_of_fs_fname ];                    */
/*FILE_NAME  ca_fname_list[ num_of_ca_fname ];                    */
  SD_END_STRUCT
  };
typedef struct fdir_req_info FDIR_REQ_INFO;
```

Fields:

filespec_pres	**SD_TRUE**. This indicates that the file specification (**fs_fname_list**) is present.
	SD_FALSE. This indicates that the file specification is not present.
cont_after_pres	**SD_TRUE**. This indicates that the continue after specification (**ca_fname_list**) is present.
	SD_FALSE. This indicates that the continue after specification is not present.
num_of_fs_fname	This contains the number of elements in the file name sequence **fs_fname_list**.
num_of_ca_fname	This contains the number of continue after elements in the **ca_fname_list**.
fs_fname_list	This structure of type **FILE_NAME** contains the file specification. Only directory entries matching this file specification are to be returned.
ca_fname_list	This structure of type **FILE_NAME** contains the continue after specification. Begin the directory response with the file after the one specified by this file name.

NOTES:

1. See the description of the file name structure on page 3-267 for more information on the format of MMS file names.
2. When allocating a data structure of type **FDIR_REQ_INFO**, enough memory must be allocated to hold the information for the file name sequences. The following C statement can be used if both **fs_fname_list** and **ca_fname_list** are to be included in the PDU.

```
info = (FDIR_REQ_INFO *) chk_malloc(sizeof(FDIR_REQ_INFO) + num_of_fs_fname *
             sizeof(FILE_NAME) + num_of_ca_fname * sizeof(FILE_NAME));
```

Response/Confirm

The operation specific data structure described below is used by the server in issuing a FileDirectory response (**mp_fdir_resp**). It is received by the client when a FileDirectory confirm (**u_mp_fdir_conf**) is received.

```
struct fdir_resp_info
  {
  ST_INT       num_dir_ent;
  ST_BOOLEAN   more_follows;
/*FDIR_DIR_ENT dir_ent_list [num_dir_ent]                          */
  SD_END_STRUCT
  };
typedef struct fdir_resp_info FDIR_RESP_INFO;
```

Fields:

num_dir_ent	This indicates the number of directory entries in this response.
more_follows	**SD_TRUE**. There are more directory entries available that cannot be sent in this response. More requests will have to be issued to obtain the entire directory.
	SD_FALSE. There are no more directory entries. This is the default.
dir_ent_list	This structure of type **FDIR_DIR_ENT** contains the list of directory entries for this response. See the next page for a description of this structure.

NOTE: When allocating a data structure of type **FDIR_RESP_INFO**, enough memory must be allocated to hold the information for the file directory entries if **num_dir_ent != 0**. The following C statement can be used:

```
info = (FDIR_RESP_INFO *) chk_malloc(sizeof(FDIR_RESP_INFO) + total_num_dir_ent *
             (sizeof(FDIR_DIR_ENT) + total_num_of_fname * sizeof (FILE_NAME)));
```

```
struct fdir_dir_ent
  {
  ST_UINT32  fsize;
  ST_BOOLEAN mtimpres;
  time_t     mtime;
  ST_INT     num_of_fname;
/*FILE_NAME  fname_list[ num_of_fname ];                          */
  SD_END_STRUCT
  };
typedef struct fdir_dir_ent FDIR_DIR_ENT;
```

Fields:

`fsize` This indicates the number of bytes in the file.

`mtimpres` **SD_FALSE**. **mtime** is not included in the PDU.

SD_TRUE. **mtime** is included in the PDU.

`mtime` This contains the time, in the C language format, the file was last modified

`num_of_fname` This indicates the number of elements in the file name sequence specifying the file name in this directory entry.

`fname_list` This list of structures of type **FILE_NAME** contains the elements that comprise the file name sequence. This specifies the name of the file in this entry.

Paired Primitive Interface Functions

mp_fdir

Usage: This primitive request function sends a FileDirectory request PDU to a remote node. It uses the data found in a structure of type **FDIR_REQ_INFO** pointed to by **info**. The FileDirectory service is used to obtain attributes (directory information) about files in the virtual filestore at the remote node.

Function Prototype:

```
MMSREQ_PEND *mp_fdir (ST_INT chan,FDIR_REQ_INFO *info);
```

Parameters:

chan	This integer contains the channel number over which the FileDirectory PDU is to be sent.
info	This pointer to a structure of type **FDIR_REQ_INFO** contains information specific to the FileDirectory PDU to be sent.

Return Value:

MMSREQ_PEND *	This pointer to the request control data structure of type **MMSREQ_PEND** is used to send the FileDirectory PDU. In case of an error, the pointer is set to null and **mms_op_err** is written with the error code.

Corresponding User Confirmation Function: **u_mp_fdir_conf**

Operation-Specific Data Structure Used: **FDIR_REQ_INFO**

See page 3-321 for a detailed description of this structure.

u_fdir_ind

Usage: This user indication function is called when a FileDirectory indication is received. The contents and operation of this function are user-defined, and depend on the application requirements. To conform to the MMS protocol, the application must do one of the following some time after the indication is received:

1) send a positive response using either:

   a) the primitive response function (**`mp_fdir_resp`**) after the directory information has been successfully obtained,

   b) the virtual machine response function (**`mv_fdir_resp`**) to have the virtual machine take the requested action and send the response, or

2) call the appropriate primitive negative (error) response function (**`mp_err_resp`**). See **Module 11 — MMS-EASE Error Handling** on page 3-343 for more information on this function.

See **Volume 1 — Module 1 — User Indication Function Class** for additional information regarding the recommended handling of indications.

Function Prototype: `ST_VOID u_fdir_ind (MMSREQ_IND *ind);`

Parameters:

`ind` This pointer to the **`MMSREQ_IND`** structure is described in **Volume 1 — Module 2 — Request and Indication Control Data Structures**. This pointer must be used to send the response to this indication. The primary item of interest in this structure is **`ind->resp_info_ptr`**. This is a pointer to the operation specific data structure for a FileDirectory indication (**`FDIR_REQ_INFO`**). This pointer will always be valid when **`u_fdir_ind`** is called.

Return Value: `ST_VOID` (ignored)

Operation-Specific Data Structure Used: **`FDIR_REQ_INFO`**

See page 3-321 for a detailed description of this structure.

mp_fdir_resp

Usage: This primitive response function sends a FileDirectory positive response PDU using the data from a structure of type **FDIR_RESP_INFO**, pointed to by **info**. This function should be called as a response to the **u_fdir_ind** function being called (a FileDirectory indication is received), and after the directory information has been successfully obtained.

Function Prototype:
```
ST_RET mp_fdir_resp (MMSREQ_IND *ind,                FDIR_RE-
                                SP_INFO *info);
```

Parameters:

`ind` This pointer to the indication control data structure of type **MMSREQ_IND** is passed to the **u_fdir_ind** function when it was called.

`info` This pointer to an Operation Specific data structure of type **FDIR_RESP_INFO** contains information specific to the response PDU to be sent.

Return Value: `ST_RET` **SD_SUCCESS**. No Error.

`<> 0` Error Code.

Corresponding User Indication Function: **u_fdir_ind**

Operation-Specific Data Structure Used: **FDIR_RESP_INFO**

See page 3-322 for a detailed description of this structure.

u_mp_fdir_conf

Usage: This primitive user confirmation function is called when a confirm to a primitive FileDirectory request (**mp_fdir**) is received. **resp_err** contains a value indicating whether an error occurred. **resp_info_ptr** may contain data that the remote node sent back in the confirmation or it may contain error information if an error occurred. If an error has occurred (**req->resp_err = CNF_RESP_OK**) FileDirectory information is available to the application using the **req->resp_info_ptr** pointer. See **Volume 1 — Module 1 — Primitive User Confirmation Function Class** for more information on the recommended handling of confirms.

Note that the function called for this event may be changed if desired; however, **u_mp_fdir_conf** is the standard default function.

Function Prototype:

```
ST_VOID u_mp_fdir_conf  (MMSREQ_PEND *req);
```

Parameters:

`req` This pointer to the **MMSREQ_PEND** structure is returned from the original **mp_fdir** request function. See **Volume 1 — Module 2 — Request and Indication Control Data Structures** for more information on this structure.

For a positive response (**req->resp_err = CNF_RESP_OK**), the request control structure member **resp_info_ptr** is a pointer to the operation specific data structure (**FDIR_RESP_INFO**) for the FileDirectory function.

For a negative response, or other errors, see **Module 11 — MMS-EASE Error Handling** starting on page 3-339 for more information.

Return Value: `ST_VOID` (ignored)

Operation-Specific Data Structure Used: **FDIR_RESP_INFO**

See page 3-322 for a detailed description of this structure.

Virtual Machine FileDirectory Operations

The following section information on how to use the virtual machine interface for the FileDirectory service.

- The FileDirectory service consists of the virtual machine function of **`mv_fdir_resp`**.

There are no virtual machine data structures.

Virtual Machine Interface Functions

mv_fdir_resp

Usage: This virtual machine function allows the user to respond to a FileDirectory indication (**`u_fdir_ind`**) without explicitly having to create an input structure of type **`FDIR_RESP_INFO`** and without having to interact directly with the operating system to get the file directory.

Function Prototype: `ST_RET mv_fdir_resp (MMSREQ_IND *ind);`

Parameters:

`ind` This pointer to a indication control data structure of the type **`MMSREQ_IND`** is received for the FileDirectory indication.

Return Value: `ST_RET` **`SD_SUCCESS`**. No Error.

`<> 0` Error Code.

1. Third Party Handling Introduction

MMS Third Party functionality is provided (if supported by the VMD) to tell the VMD to obtain the file from another node on the network using some protocol (which may or may not be MMS). To support Third Party features, MMS-EASE provides a set of data structures and APIs to make the translation from an AR Name, a null-terminated ASCII string, to an ASN.1 encoded Application Reference.

An example of this follows:

> When an application program calls **`mv_init`**, it passes the arguments **`chan`** and **`partner`**. The argument, **`partner`**, is really an AR Name that MMS-EASE handles as an ASCII string. However, some services such as **`mp_obtfile`** deal with what is called an Application Reference for THIRD PARTY connection. It comes to the user in an ASN.1 encoded form. This results in the problem of not having a good way of translating an ASN.1 encoded Application Reference back to an ASCII printable AR Name needed for a call to **`mv_init`**.

To understand how MMS-EASE handles the translation problem, several terms need to be explained:

1. **AR Name** — This is a user-friendly ASCII string that refers to an explicit Application Entity name. See **Volume 1 — Module 1 — Network Addressing Issues** for further explanation of AR Name.
2. **Application Reference structure** — This is the Application Entity Name. This is in a C Language format that can be understood and dealt with by application programs.
3. **Encoded Application Reference** — This is an ASN.1 encoding for an Explicit Application Entity name. This must be supplied for some MMS PDUs. Some MMS services present this as an Encoded Application Reference.

Data Structures Used In Third Party Handling

ISO Form Data Structures

The following two data structures describe the forms that can be used to define Application References. At this time, it is necessary to understand that:

1. MMS Implementation agreements specify the use of Form 2.
2. MMS-EASE recommends that Form 1 not be used. At this time, most manufacturers supplying an ACSE stack do not support Form 1. If using Form 1, there may interoperability problems at the remote node.

ISO ASN.1 AP Title Structure (Form 1)

The structure described below is used to define one of two forms used in various components of ACSE addressing. However, this form is not supported by the MAP specification.

```
struct form1
  {
  ST_UCHAR  *ptr;
  ST_INT16  len;
  };
typedef struct form1 FORM1_T;
```

Fields:

`ptr` This is a pointer to the ASN.1 form 1 ACSE AP Title specified by ISO 8650.

`len` This is the length (in bytes) of the data pointed to by **`ptr`**.

ISO Object Identifier Structure (Form 2)

AP Titles are represented by an object identifier for OSI networks. For a thorough description of the object identifier, and its application, see **Volume 1 — Module 1 — Network Addressing** for more information. The following structure is used by MMS-EASE to represent an object identifier. This structure is designated as Form 2. It is currently supported by the MMS specification for Application References.

```
struct mms_obj_id
  {
  ST_INT   num_comps;
  ST_INT16 comps[MAX_OBJID_COMPONENTS];
  SD_END_STRUCT
  };
typedef struct mms_obj_id MMS_OBJ_ID;
```

Fields:

`num_comps`	This is the number of components actually used by this object identifier.
`comps`	This array of integers contains the individual components of the object identifier. **`MAX_OBJID_COMPONENTS`** is set by default to 16.

2. Application Reference Name Manipulation

To handle the conversion problem, MMS-EASE has developed several structures and functions to handle translating between AR Names and ASN.1. This translation is accomplished through an intermediary structure in the following manner:

AR NAME ⇔ `struct app_ref` ⇔ ASN.1 encoded Application Reference

Application Reference Data Structure

The following structure is used to define an intermediate Application Reference that consists of an AE Title and all its component parts. This structure is used in various MMS-EASE LLP request functions to provide the translation between an AR NAME and an ASN.1 encoded Application Reference. See starting on page 3-333 for explanations of the functions provided.

```
struct app_ref
  {
  ST_INT16     form;
  ST_BOOLEAN  ap_title_pres;
  union
    {
    struct     form1            form_1;
    struct    mms_obj_id  form_2;
    } ap_title;
  ST_BOOLEAN  ap_invoke_pres;
  ST_INT32     ap_invoke;
  ST_BOOLEAN  ae_qual_pres;
  union
    {
    struct     form1            form_1;
    ST_INT32  form_2;
    } ae_qual;
  ST_BOOLEAN  ae_invoke_pres;
  ST_INT32     ae_invoke;
  };
typedef struct app_ref APP_REF;
```

Fields:

`form` This indicates the form to be used for the **APP_REF** structure:

- **1** form 1, the application reference is in ASN.1 form. This form is currently not used.
- **2** form 2, the application reference is in object identifier form. It consists of members of type **MMS_OBJ_ID** and long integers. MMS-EASE currently defaults to this form.

`ap_title_pres`

SD_FALSE. AP Title is NOT included in this application reference.

SD_TRUE. AP Title is included in this application reference.

`ap_title`

form_1. This is a structure of type **form1** containing the AP Title in ASN.1 form.

form_2. This structure of type **MMS_OBJ_ID** contains the AP Title in object identifier form.

`ap_invoke_pres`

SD_FALSE. AP Invoke ID is NOT included in this application reference.

SD_TRUE. AP Invoke ID is included in this application reference.

ap_invoke	This contains the AP Invoke ID. Used only if **ap_invoke_pres = SD_TRUE**.
ae_qual_pres	**SD_FALSE**. AE Qualifier is NOT included in this application reference.
	SD_TRUE. AE Qualifier is included in this application reference.
ae_qual	**form_1**. This structure of type **FORM1** contains the AE Qualifier in ASN.1 form.
	form_2. This contains the AE Qualifier in long word form.
ae_invoke_pres	**SD_FALSE**. AE Invoke ID is NOT included in this application reference.
	SD_TRUE. AE Invoke ID is included in this application reference.
ae_invoke	This contains the AE Invoke ID. Used only if **ae_invoke_pres = SD_TRUE**.

NOTE: Please refer to **Volume 1 — Module 1 — Network Addressing** for an explanation of AP Title, AP Invoke ID, AE Qualifier, and AE Invoke ID. See page 3-330 for an explanation of the **MMS_OBJ_ID** data structure.

Application Reference Manipulation Functions

These support functions are provided in order to handle the translation between AR Names and ASN.1.

ms_appref_to_arname

Usage: This function translates an intermediate form of an Application Reference (of structure type **APP_REF**) to an AR Name that is a null-terminated character string. This function is successful only if the information on the **APP_REF** structure matches the information in the Local DIB.

Function Prototype:

```
ST_RET ms_appref_to_arname (ST_CHAR *arname,
                            APP_REF *appref);
```

Parameters:

`arname` This pointer to a character string contains the AR Name into which the translated **APP_REF** will be placed.

`appref` This structure of type **APP_REF** points to the address of the AE Title to be translated. See page 3-331 for a detailed description of this structure.

Return Value: `ST_RET`

SD_SUCCESS. No error. Application Reference translated to an AR Name.

SD_FAILURE. Error. Application Reference not translated to an AR Name.

NOTE: The **arname** must be a pointer to a place in memory big enough to hold the **ar_name[MAX_AR_LEN+1]**, where **MAX_AR_LEN = 64**.

ms_appref_to_asn1

Usage: This function translates an intermediate form of an Application Reference (of structure type **APP_REF**) to an ASN.1 encoded string. This ASN.1 string can be used as input or output parameters to some MMS-EASE service primitives. This function only supports form 2 for this structure. This is an AE Title whose members can consist of AP Title of structure type **MMS_OBJ_ID** and AP Invoke ID, AE Qualifier, and AE Invoke IDs, which are LONG integers.

Function Prototype:

```
ST_RET ms_appref_to_asn1 (APP_REF *appref,
                          ST_UCHAR *dest,
                          ST_INT dest_len,
                          ST_UCHAR **asn1_out,
                          ST_INT *asn1_len_out);
```

Parameters:

`appref` This structure of type **APP_REF** points to the Application reference to be translated. See page 3-331 for a detailed description of this structure.

`dest` This pointer to the buffer contains the ASN.1 encoded string of the Application Reference to be translated.

`dest_len` This is the length, in bytes, of the buffer pointed to by **dest**.

`asn1_out` This points to the address returned after the ASN.1 string was encoded from the Application Reference.

`asn1_len_out` This is the pointer to the actual length of the encoded ASN.1 string.

Return Value: `ST_RET`

SD_SUCCESS. No error. Application Reference translated to an ASN.1 string.

SD_FAILURE. Error. Application Reference not translated to an ASN.1 string.

ms_arname_to_appref

Usage: This function is used to translate an AR Name. This is a null-terminated character string, to an intermediate form of an Application Reference (of structure type **APP_REF**). This function only supports form 2 for this structure. This is an AE Title whose members can consist of AP Title of structure type **MMS_OBJ_ID** and AP Invoke ID, AE Qualifier, and AE Invoke IDs which are long integers.

Function Prototype:

```
ST_RET ms_arname_to_appref (ST_CHAR *arname,
                            APP_REF **appref_out);
```

Parameters:

`arname` This pointer to a character string contains the AR Name to be converted, and placed in the **APP_REF** structure.

`appref_out` This pointer is a structure of type **APP_REF** points to the address of the converted Application Reference. See page 3-331 for a detailed description of this structure.

Return Value: `ST_RET`

SD_SUCCESS. No error. AR Name translated to an Application Reference.

SD_FAILURE. Error. AR Name not translated to an Application Reference.

ms_arname_to_asn1

Usage: This function translates an AR Name to corresponding ASN.1 encoded equivalent of the Application Reference. This ASN.1 string can be used as input or output parameters to some MMS-EASE service primitives.

Function Prototype:

```
ST_RET ms_arname_to_asn1 (ST_CHAR *arname,
                          ST_UCHAR *asn1_buf,
                          ST_INT asn1_buf_len,
                          ST_UCHAR **asn1_out,
                          ST_INT *asn1_len_out);
```

Parameters:

`arname` This pointer to the null-terminated character string represents the AR Name to be translated.

`asn1_buf` This pointer to the buffer contains the ASN.1 encoded string of the Application Reference to be converted.

`asn1_buf_len` This is the maximum length, in bytes, of the build buffer pointed to by **`asn1_buf`**.

`asn1_out` This pointer to the address is returned after the ASN.1 string was encoded from the AR Name.

`asn1_len_out` This is the pointer to the actual length of the encoded ASN.1 string.

Return Value: `ST_RET`

`SD_SUCCESS`. No error. AR Name translated to an ASN.1 string.

`SD_FAILURE`. Error. AR Name not translated to an ASN.1 string.

ms_asn1_to_appref

Usage: This function translates an ASN.1 encoded version of an Application Reference to an intermediate form of an Application Reference (of structure type **APP_REF**). This function only supports form 2 for this structure. This is an AE Title whose members can consist of AP Title of structure type **MMS_OBJ_ID** and AP Invoke ID, AE Qualifier, and AE Invoke IDs which are LONG integers.

Function Prototype:

```
ST_RET ms_asn1_to_appref (APP_REF *appref,
                          ST_UCHAR *asn1,
                          ST_INT asn1len);
```

Parameters:

appref	This structure of type **APP_REF** points to where the translated intermediate Application Reference is to be placed. See page 3-331 for a detailed description of this structure.
asn1	This pointer to the buffer stores the ASN.1 encoded string to be translated.
asn1len	This is the maximum length, in bytes, of the buffer pointed to by **asn1** containing the ASN.1 encoded string.

Return Value:

ST_RET	**SD_SUCCESS**. No error. ASN.1 string translated and placed in structure of type **APP_REF**.
	SD_FAILURE. Error. ASN.1 string not translated and placed in structure of type **APP_REF**.

ms_asn1_to_arname

Usage: This function translates an ASN.1 encoded string to an AR Name. This is a null-terminated character string representation of an Application Reference.

Function Prototype:

```
ST_RET ms_asn1_to_arname (ST_CHAR *arname,
                          ST_UCHAR *asn1,
                          ST_INT asn1len);
```

Parameters:

`arname`	This pointer to the null-terminated character string contains the AR Name.
`asn1`	This pointer to the buffer contains the ASN.1 encoded string to be translated.
`asn1len`	This is length, in bytes, of the data pointed to by **`*asn1`**.

Return Value: `ST_RET`

`SD_SUCCESS`. No error. ASN.1 encoded string translated to an AR Name.

`SD_FAILURE`. Error. ASN.1 encoded string not translated to an AR Name.

1. MMS-EASE Error Handling

MMS-EASE error handling falls into three basic classes:

1. **Function execution errors** such as when the user program attempts to issue a read request when no association has been established using initiate.
2. **Protocol errors** such as when MMS-EASE is unable to parse correctly an incoming indication.
3. **Application errors** such as when a remote server application was unable to carry out a request and returned a negative response.

Function Execution Errors

Most MMS-EASE functions accessible to the user application provide an indication that the function was not executed correctly. For functions that return pointers, a pointer value of 0 (zero) or null indicates an execution error. When this occurs, the global MMS-EASE integer variable **`mms_op_err`** is written with the error code. For functions that do not return pointers, the return value is an error flag or code. If the return value is **`SD_SUCCESS`** (zero), it means the function executed correctly; if not 0, then the return value is the error code.

Even if the MMS-EASE return code indicates success, the OSI Stack may later reject the operation before sending it to the lower layers of the network software. If this happens on a request function, the response will never be received. You would not get a transport timeout because transport never got the message. If this happens, MMS-EASE automatically calls one of the exception indication functions. Because of the seriousness of this type of error, you may want to abort the association at this time.

The exception indication function for errors related to the Lower Layer Provider (LLP) is named **`u_llp_error_ind`**. The MMS exception indication function for all other errors is named **`u_mmsexcept_ind`**. Typically, if an error is received from ACSE, it is related to the association state or an OSI stack problem. If an exception is received from MMS, it is typically something related to a MMS state problem. Again, if an exception indication is received using any of these functions, it is advisable to abort any active association before retrying any communications on that channel. MMS-EASE error code meanings are documented in **Volume 1 — Appendix A**.

Protocol Errors

MMS-EASE protocol errors typically consist of such things as PDU parse errors, and unsupported service requests. When a PDU parse error occurs on an indication, a reject response is sent automatically. If the PDU parse error occurs in a confirmation, MMS-EASE tries to match the confirmation to an outstanding request, and sets the response error code for that request. If a PDU cannot be parsed at all, the PDU is ignored. The functions, **`mp_reject_conf`** and **`mp_reject_ind`**, are available for explicitly rejecting an indication or confirm that MMS-EASE did not reject.

Application Errors

Error responses may need to be sent to indications received because the requested action cannot be performed, an error occurred while trying to implement the requested action, or some or all of the data was invalid. To issue an error response to an indication, simply call the appropriate primitive error response function (**mp_err_resp**, **mp_cancel_err**, **mp_init_err**, or **mp_conclude_err**). If using the virtual machine to handle responses, these error functions will not need to be called, as error responses are automatically generated as needed by the virtual machine.

Conversely, the remote application may not be able to carry out a request from the local application due to such issues as a lack of resources, unrecognized object name, or a failure to perform the requested action. In this case, a negative confirm will be received, and **req->resp_err** will be equal to **ERR_OK** in the local application's user confirm function (**u_mp_xxxx_conf** or **u_mv_xxxx_conf**). The remainder of this section deals with application errors.

Error Information Structure

When the user confirmation function (**u_mp_xxxx_conf** or **u_mv_xxxx_conf**) is called, and the pending request data structure (**MMSREQ_PEND**) indicates an application error occurred (**req->resp_err==ERR_OK**), the error information is placed in the error information structure pointed to by the **resp_info_ptr** member of the **MMSREQ_PEND** structure. This is passed to the **u_mp_xxxx_conf** or **u_mv_xxxx_conf** function. This structure can then be examined to see what type of error occurred. The error information structure is of the following form:

```
struct err_info
  {
  ST_INT16            eclass;
  ST_INT16            code;
  ADTNL_ERR_RESP_INFO adtnl;
  };
typedef struct err_info   ERR_INFO;
```

Fields:

`eclass` This is the error class indicating the class of the error that occurred, per ISO 9506. The class of the error indicates to which type of service the error corresponds. For each class of error, there are different meanings for the codes. For instance, the same **code** number will have different meanings when the error **eclass** differs. See **Volume 1 — Appendix A** for a list of error class and codes.

`code` This variable contains the code indicating the specific reason that the service was not executed corresponding to the specified **eclass**, per ISO 9506.

`adtnl` This structure of type **ADTNL_ERR_RESP_INFO** contains additional error information. This structure is documented on the next page. Some companion standards may require that the additional error response information be used. Some services such as ObtainFile, FileRename, Start, Stop, Resume, and Reset require this additional information as well.

```
struct adtnl_err_resp_info
  {
  ST_BOOLEAN  mod_pos_pres;
  ST_INT32    mod_pos;
  ST_BOOLEAN  info_pres;
  ST_BOOLEAN  code_pres;
  ST_INT32    code;
  ST_BOOLEAN  descr_pres;
  ST_CHAR     *descr;
  ST_BOOLEAN  ssi_pres;
  ST_INT16    service;
  ST_UINT32   ss_error_val;
  OBJECT_NAME ss_error_oname;
  ST_INT      ss_error_len;
  ST_UCHAR    *ss_error_data;
  };
typedef struct adtnl_err_resp_info ADTNL_ERR_RESP_INFO;
```

Fields:

`mod_pos_pres` **SD_FALSE**. The **mod_pos** member is not active.

SD_TRUE. The **mod_pos** member is present.

`mod_pos` This indicates the position of the modifier to which the error pertains.

`info_pres` **SD_FALSE**. There is no useful information in this structure. In this case, the user application need not look at any other members of this structure, as they will all be "don't care."

SD_TRUE. There is information present in this structure.

`code_pres` **SD_FALSE**. The **code** member is not active.

SD_TRUE. The **code** member is present. It includes a non-standard, user-defined error code.

`code` This contains an additional implementation-specific code.

`descr_pres` **SD_FALSE**. The **descr** member is not active.

SD_TRUE. The **descr** member is present.

`descr` This is a pointer to additional description of the error. This should be a printable, human readable character string containing implementation- specific information regarding the error.

`ssi_pres` **SD_FALSE**. The **service**, **ss_error_len,** and **ss_error_data** members are not active. **S**ervice **S**pecific **I**nformation (SSI) is not present.

SD_TRUE. The **service** member is present and the **ss_error_len**, and **ss_error_data** members may be present. Service Specific information (ssi) is present.

`service` This indicates the type of service-specific error information contained in **`ss_error_val`** or **`ss_error_data`**:

- **`0`** Obtain File-Error (use **`ss_error_val`**)
- **`1`** Start-Error (use **`ss_error_val`**)
- **`2`** Stop-Error (use **`ss_error_val`**)
- **`3`** Resume-Error (use **`ss_error_val`**)
- **`4`** Reset-Error (use **`ss_error_val`**)
- **`5`** DeleteVariable Access-Error (use **`ss_error_val`**)
- **`6`** DeleteNamed Variable List-Error (use **`ss_error_val`**)
- **`7`** DeleteNamed Type-Error (use **`ss_error_val`**)
- **`8`** DefineEventEnrollment-Error (use **`ss_error_val`**)
- **`9`** File Rename-Error (use **`ss_error_val`**)
- **`10`** AdditionalService-Error. Examine data pointed to by **`ss_error_data`** for the service specific error information.

`ss_error_val` This indicates a service specific error value when **`service`** is equal to 0-7 or 9 (IS) or a service specific object name when **`service`** is equal to 8 (IS-Define Event Enrollment).

`ss_error_oname` This indicates a service specific additional service error when **`service`** is equal to **`10`** (IS-Companion Standard).

`ss_error_len` This indicates the length, in bytes, of the data pointed to by **`ss_error_data`**. This is only used if **`service = 10`**.

`ss_error_data` This pointer to specific error information conforms to the companion standard governing the particular VMD that the error response is from. Refer to the specific companion standard governing the particular VMD from which the error response was received.

2. Error Handling Functions

This section describes the various error handling functions used by MMS-EASE.

mp_err_resp

Usage: This function is used to send an error response (result(-)) PDU. It is used to send a negative response (result(-)) to respond negatively to any other confirmed service request except for Cancel, Conclude, and Initiate. See **Volume 1 — Module 3 —** **`mp_cancel_err`**, **`mp_conclude_err`**, **`mp_init_err`** for more information. This function should be called in place of the positive response function (see any **`mp_xxxx_resp`** function) if the requested service cannot be performed. This function should also be called if a service was canceled successfully.

Please refer to **Volume 1 — Appendix A** for a complete description of the various error classes and codes specified by MMS for use in the **`err_class`** and **`code`** inputs. Be sure to fill in the **`ADTNL_ERR_INFO`** structure before calling **`mp_err_resp`**.

Function Prototype:

```
ST_RET  mp_err_resp (MMSREQ_IND *ind,
                     ST_INT16 err_class,
                     ST_INT16 code);
```

Parameters:

`ind` This pointer to the indication control data structure of type **`MMSREQ_IND`** corresponds to the service request that was not performed successfully. This is the same pointer passed to the **`u_xxxx_ind`** function when it was called.

`err_class` This integer contains the particular class of the error per ISO 9506. See **Volume 1 — Appendix A**.

`code` This integer contains the code indicating the specific reason that the service was not executed corresponding to the specified **`err_class`**, per ISO 9506. See **Volume 1 — Appendix A**.

Return Value: `ST_RET`

`SD_SUCCESS`. No Error.

`SD_FAILURE`. Error (error response not sent).

. . . .continued on following page. . . .

mp_err_resp . . . .continued from preceding page. . . .

Other Data Structures Used: **ADTNL_ERR_INFO**

```
extern ADTNL_ERR_RESP_INFO  adtnl_err_info;
```

The **ADTNL_ERR_INFO** structure is a global non-allocated structure of type **ADTNL_ERR_RESP_INFO**. It is used to hold the additional error information for the next call to **mp_err_resp**. Additional error response information may be included in your error PDU. For some services this additional information is optional. It can contain some human-readable or other information. Whether additional error response information is to be included or not, the following steps must be performed to send an error response:

1. Fill out the **ADTNL_ERR_INFO** structure with the required information. At a minimum, **adtnl_err_info.info_pres** must be set to zero to indicate that no additional error information is present.
2. Call the **mp_err_resp** function with the appropriate **err_class**, **code**, and ***ind**.

Please refer to page 3-325 for more information concerning the **ADTNL_ERR_RESP_INFO** and **ADTNL_ERR_INFO** structures.

NOTES:

1. Normally MMS-EASE automatically sends error responses for service requests that are canceled before **u_xxxx_ind** is called. MMS-EASE also will automatically send the appropriate Cancel Response PDU in this case.
2. Please refer to ISO 9506 or **Volume 1 — Appendix A** for the allowed values of the error **err_class** and **code**. MMS-EASE does not perform any type checking on the error codes supplied here. It is the application program's responsibility to make sure that the error class and code are correct for the given service and error condition.
3. The additional error response information (**ADTNL_ERR_INFO**) is MANDATORY for the following services:

   Start, Stop, Resume, Reset. See **Volume 2 — Module 6 — Program Invocation Management**.

   DeleteVariableAccess, DeleteNamedVariableList, DeleteNamedType. See **Volume 2 — Module 5 — Variable Access and Management**.

   ObtainFile, FileRename. See **Module 9 — File Access and Management** starting on page 3-255.

u_mmsexcept_ind

Usage: This function is called by MMS-EASE when an exception condition has occurred. This is an error generally from which MMS-EASE cannot recover. This function should be used to examine error information and take any steps appropriate for your application. This typically means that MMS-EASE is unusable and must be re-started. It is recommended to log the exception condition to determine what caused the exception to occur. See **Volume 1 — Appendix A** for a description of the various exception conditions that MMS-EASE detects.

Function Prototype: `ST_VOID u_mmsexcept_ind (ST_INT chan, ST_RET code);`

Parameters:

`chan` This is the channel number over which the exception condition was detected. A value of -1 for chan indicates a general exception that does not apply to any particular channel such as the LLP running out of memory resources.

`code` This value indicates the type of MMS-EASE exception that has occurred. See **Volume 1 — Appendix A** for more information on the codes that can appear for this variable and their meaning.

Return Value: `ST_VOID` (ignored)

The support functions described on the following pages deal with logging and displaying error codes.

ms_perror

Usage: This function is a diagnostic function that displays the description to a corresponding error code. All MMS, Virtual Machine Interface, LLP, and stack or driver error descriptions are produced from this function. If an invalid or undescribed error code is used as an argument, the following description will be logged:

```
error code 'error' not found
```

Function Prototype: `ST_VOID ms_perror (ST_RET code);`

Parameters:

`code` This is an error code found in the global variable **`mms_op_err`** returned from a MMS-EASE function. See **Volume 1 — Appendix A** for a complete list of error codes.

Return Value: `ST_VOID` (ignored)

ms_perror_log

Usage: This function maps an error code to a textual description of the error. The description is further directed to the specified file pointer. In MMS-EASE, **ms_perror** calls this function with the **fp** argument of **stdout**. This indicates standard output that defaults to a terminal screen.

Function Prototype:

```
ST_VOID ms_perror_log (FILE *fp, ST_RET code);
```

Parameters:

fp	This is a pointer to the file of structure type **FILE** where the logging of error descriptions are to be stored.
code	This is an error code found in the global variable **mms_op_err** or returned from a MMS-EASE function. See **Volume 1 — Appendix A** for a complete list of error codes.

Return Value: ST_VOID (ignored)

ms_perror_str

Usage: This function allows the capability of printing a MMS-EASE error code description to a buffer rather than a file stream. This may be useful in implementing application logging systems.

Function Prototype:

```
ST_VOID ms_perror_str (ST_CHAR *dest,
                       ST_INT dest_len,
                       ST_RET code);
```

Parameters:

`dest`	This is a pointer to destination buffer in which to put the error string.
`dest_len;`	This indicates the maximum length of the destination buffer pointed to by **`dest`**.
`code`	This is an error code found in the global variable **`mms_op_err`** or returned from a MMS-EASE function. See **Volume 1 — Appendix A** for a complete list of error codes.

Return Value: `ST_VOID` (ignored)

mms_err_rsp

Usage: This function is called to decode an error response PDU.

Function Prototype: `ST_VOID mms_err_rsp (ST_VOID);`

Parameters: NONE

Return Value: `ST_VOID` (ignored)

Appendix A. Error Codes

Error and exception codes are represented by two-byte integer values. The high byte defines the entity that generated the code, the low byte is the actual code:

High byte Entity code	Low byte Error code

The entity part of the error (exception code) and the actual code (info) are defined using hexadecimal representations. The following is a list of the defined MMS-EASE error codes. The Entity Codes are defined using the tables found in the file, **glberror.h**. The actual Error Code is an offset into these tables specified by the Entity Code. Most of these offsets are defined in the file, **mms_perr.h**. The function **ms_perror** examines these codes, and returns a description. See **Volume 3 — Module 11 — Error Handling** for an explanation of this function.

These error codes are valid for functions that return **ST_RET** where a non-zero value indicates an error. In addition, functions that return pointers will set a null pointer when an error is detected. The value of the global integer variable **mms_op_err** contains the error code.

All MMS-EASE, SUIC error codes are described as follows. Shown in *italics* are the actual descriptions returned from **ms_perror**.

The following codes may be returned from any request or response function.

DEFINED CONSTANT	HEX VALUE	MEANING
ERROR CODES FOR GENERAL PROVIDER: E_MMS_GEN 0x6400		
ME_CHAN_STATE	0x6401	*Channel State* Association state is not OK for the operation. Check channel state for proper state (e.g., must be IDLE for Initiate).
ME_CHAN_NUM	0x6402	*Channel Number* Bad Channel Number. Channel number is not valid.
ME_TITLE_LEN	0x6403	*AR Name Len* Maximum length of AR Name (local or remote) exceeds value of **MAX_AR_LEN** (by default set to 64).
ME_SEND_SIZE	0x6404	*Send Size* Size of message received in **u_mv_fread_conf** (Remote FileRead) is larger than maximum message size.
ME_REQ_PEND_COUNT	0x6405	*Too Many Outstanding Requests* There are too many requests pending on a channel.
ME_INACTIVE_IND	0x6406	*Indication Control Pointer Not Active* The specified indication is not active or the indication control structure is invalid.
ME_PARTNER_NAME	0x6407	*Remote AR Name Unknown* The partner (remote) AR name is not known. Please check configuration file (DIB).
ME_AR_NAME	0x6408	*Local AR Name Unknown* The local AR name is not known. Please check the configuration file (DIB).

DEFINED CONSTANT	HEX VALUE	MEANING
ME_OBJ_ID_ERR	0x6409	*ObjectID* Invalid Object Identifier encountered.
ME_SEND_ERROR	0x640A	*PDU Send* Error sending a cancel request PDU.
ME_ACSE_CHAN_RANGE	0x640B	*ACSE Channel Range* ACSE Channel Range exceeds **MAX_MMS_CHAN** (default is 4).
ME_LLC_CHAN_RANGE	0x640D	*LLC Channel Range* LLC Channel Range exceeds **MAX_MMS_CHAN** (default is 4).
ME_LLP_TYPE	0x640E	*LLP Type* Incorrect LLP Type passed on association.
ME_TRANS_ID_UNAVAILABLE	0x640F	*No Trans ID Available* Transaction ID not available — LLC error only.
ME_CHAN_TYPE	0x6410	*Chan Type* Channel Type incorrect — LLC error only.
ME_TOO_MANY_CONTEXTS	0x6411	*Too Many P-Contexts* There are too many Presentation Contexts defined. See **Volume 1 — Module 3 — Context Management** for more information on P-Contexts.
ME_UNKNOWN_P_CONTEXT	0x6412	*Unknown P-Context* Attempting to send an undefined P-Context. See **Volume 1 — Module 3 — Context Management** for more information.
ME_P_CONTEXT_ERROR	0x6413	*P-Context Error* Error in sending Presentation Context — invalid P-Context. See **Volume 1 — Module 3 — Context Management** for more information.
ME_ASN1_ENCODE_OVERRUN	0x6414	*ASN1 Encode Buffer Overrun* ASN.1 Encode buffer size has been exceeded.

These codes are returned from **mp_xxxx** functions**:**

ERROR CODES FOR PAIRED PRIMITIVE INTERFACE: E_MMS_PPI 0x6500

ME_QUEFULL	0x6501	*No Request Control Structures Available* MMS-EASE internal queue overflow. There are too many outstanding requests or indications.

The following error codes are valid in the **resp_err** member of a **MMSREQ_PEND** queue data structure used to send out a request. The error codes are valid within the **u_mp_xxxx_conf** and **u_mv_xxxx_conf** functions**.**

ERROR CODES FOR REQUEST CONTROL BLOCK: E_MMS_RESP 0x6600

CNF_PARSE_ERR	0x6601	*Confirm PDU Parse Error* Confirm PDU Parse error is found locally.
CNF_REJ_ERR	0x6602	*Peer Rejected* Request rejected by peer.
CNF_ERR_OK	0x6603	*Error Response* Error response received. **resp_info_ptr** points to **err_info**.
CNF_DISCONNECTED	0x6604	*Connection Terminated* Association disconnected or aborted.

DEFINED CONSTANT	HEX VALUE	MEANING
CNF_CHAN_OP_ERR	0x6605	*Opcode/Invoke ID Mismatch* Wrong operation for specified Invoke ID.
CNF_CANST_ERR	0x6606	*Cancel State* Cancel state error — cannot cancel.
CNF_ASS_REQ_REJECTED	0x6607	*Associate Request Rejected* Initiate request rejected.
CNF_ASS_RESP_PARAM	0x6608	*Associate Response Parameters* Invalid Initiate confirm received.
CNF_ASS_USER_REJ_CONF	0x6609	*User Rejected Associate Confirm* Initiate confirm rejected locally by user.
CNF_REM_FOPEN	0x660A	*Remote File Open* Cannot open remote file — possibly attempting to open a remote file that is already open or is non-existent.
CNF_INIT_PARAM	0x660B	*Initiate Parameter* Initiate parameter problem — invalid negotiated parameters.
CNF_REM_FREAD	0x660C	*Remote File Read* Error reading remote file.
CNF_LOC_FWRITE	0x660D	*Local File Write* Error writing data to local file.
CNF_REM_FCLOSE	0x660E	*Remote File Close* Error closing remote file.
CNF_LOC_FCLOSE	0x660F	*Local File Close* Error closing local file.
CNF_MVREAD_RESP_PARAM	0x6610	*Read Response Parameter* Invalid read response parameter for **mv_read_resp**.
CNF_VM_RESP_ERR	0x6611	*Could Not Send Response* Error in sending Virtual Machine download response.
CNF_LLC_SEND_ERROR	0x6612	*LLC Send Error* Error sending LLC request.

These codes are returned from **mv_xxxx_resp** functions:

ERROR CODES FOR VIRTUAL MACHINE INTERFACE: E_MMS_VMI 0x6700		
MVE_VARNAME	0x6701	*Variable Name* Problem with a variable name such as variable name not defined.
MVE_TYPENAME	0x6702	*Type Name* Problem with a type name such as type name not defined.
MVE_ADDR	0x6703	*Address* Variable address problem — unknown type of address.
MVE_FOPEN	0x6704	*File Open* Problem opening file.
MVE_REM_FILE_COUNT	0x6705	*Too Many Remote Files Open* There are too many remote files open, exceeded maximum set by variable **MAXFILESOPEN** (default of 20 total files, local + remote).

DEFINED CONSTANT	HEX VALUE	MEANING
MVE_LOC_FILE_COUNT	0x6706	*Too Many Local Files Open* There are too many local files open, exceeded maximum set by variable **MAXFILESOPEN** (default of 20 total files, local + remote).
MVE_FILE_REFNUM	0x6707	*File Reference Not Found (frsmid)* The File Reference State Machine ID (FRSMID) is unknown.
MVE_FRENAME	0x6708	*File Rename* Problem in renaming a file.
MVE_FDELETE	0x6709	*File Delete* Problem in deleting a file.
MVE_FDIR	0x670A	*File Directory* Problem in obtaining a file directory.
MVE_LOC_FOPEN	0x670B	*Local File Open* Attempting to open a local file already open.
MVE_CANCEL_STATE	0x670C	*Cancel State* Current state won't allow canceling a request.
MVE_DOM_ERR	0x670D	*Domain Type Error* Invalid domain specified in upload/download or invalid domain-specific type or variable.
MVE_RT_TYPE	0x670E	*Runtime Type* Error in converting to runtime type.
MVE_DOM_STATE	0x670F	*Domain State* Domain is not in the proper state to be downloaded or deleted (e.g., Domain is being used).
MVE_AT_UPPER_LIMIT	0x6710	*Too Many Items* Some defined MMS object has exceeded the maximum number — check **max_mmsease_doms**, **max_mmsease_pis**, **max_mmsease_types**, **max_mmsease_vars**.
MVE_DATA_CONVERT	0x6711	*Data Conversion* Error in converting data.
MVE_VM_SERVICE_NOTSUPP	0x6712	*Unsupported Indication Parameter* Request not supported by virtual machine. Check **mmsop_en.h** to make sure service is enabled.
MVE_MVWRITE_REQ_PARAM	0x6713	*Indication Parameter Problem* Invalid parameter for **mv_write**. Number of variables is not equal to the number of data.
MVE_DOMAIN_NAME	0x6714	*Domain Name* Invalid domain name specified (e.g., domain name is not defined).
MVE_OBJECT_SCOPE	0x6715	*Invalid Object Scope* MMS object scope invalid.
MVE_VAR_LIST	0x6716	*Variable List Not Supported* In the **ms_extract_varname** function, asking for a named variable list rather than a list of variables.
MVE_VAR_NUMBER	0x6717	*Variable Select Index* In the **ms_extract_varname** function, index is too big — asking for too many variables.

DEFINED CONSTANT	HEX VALUE	MEANING
MVE_NOT_NAMED_VAR	0x6718	*Not Named Variable* In the **ms_extract_varname** function, asking for a variable that is not a named variable.
MVE_WRONG_OP	0x6719	*MMS Opcode* In the **ms_extract_varname** function, asking for a wrong operation code for this function (valid are MMSOP_READ, MMSOP_WRITE, or MMSOP_INFO_RPT).
MVE_MVINFO_ACCESS_ERR	0x671A	*Access Result = Failure* In the **ms_extract_info_name** function, the remote node (Peer) could not access variable's data.
These codes are returned from the **ms_mk_asn1_type** function:		
MVE_TYPEDEF_LEN0	0x671B	*Empty Type Def'n String* Type definition string has a length of 0.
MVE_TYPEDEF_SYM_GT	0x671C	*'>' symbol* '>' symbol out of place in type definition string.
MVE_TYPEDEF_SYM_LT	0x671D	*'<' symbol* '<' symbol out of place in type definition string.
MVE_TYPEDEF_OBJNAME	0x671E	*invalid object name* Object name used for type is invalid.
MVE_TYPEDEF_SYM_RBRACE	0x671F	*'}' symbol* '}' symbol out of place in type definition string.
MVE_TYPEDEF_SYM_LBRACE	0x6720	*'{' symbol* '{' symbol out of place in type definition string.
MVE_TYPEDEF_SYM_COMMA	0x6721	*',' symbol* ',' symbol out of place in type definition string.
MVE_TYPEDEF_SYM_RPAREN	0x6722	*')' symbol* ')' symbol out of place in type definition string.
MVE_TYPEDEF_SYM_LPAREN	0x6723	*'(' symbol* '(' symbol out of place in type definition string.
MVE_TYPEDEF_SYM_P	0x6724	*'p' encountered twice* Duplicate 'p' in type definition string.
MVE_TYPEDEF_SYM_PLUS	0x6725	*'+' symbol* '+' symbol out of place in type definition string.
MVE_TYPEDEF_SYM_MINUS	0x6726	*'-' symbol* '-' symbol out of place in type definition string.
MVE_TYPEDEF_SYM_T	0x6727	*'t' symbol* 't' symbol out of place in type definition string.
MVE_TYPEDEF_BADLEN3	0x6728	*bad length for types 3 & 11* Bad type length for Bool or Gtime types.
MVE_TYPEDEF_BADLEN4	0x6729	*bad length for types 4 & 9* Bad type length for Bstring or Ostring types.
MVE_TYPEDEF_BADLEN5	0x672A	*bad len for types 5,6,12,13* Bad type length for integer, BCD, or Btime types.
MVE_TYPEDEF_BADLEN7	0x672B	*bad type length for type 7* Bad type length for Float or Double types.
MVE_TYPEDEF_BADTYPE	0x672C	*invalid prim type !(>3 && <13)* Type not defined.

DEFINED CONSTANT	HEX VALUE	MEANING
MVE_TYPEDEF_SYM_COLON	0x672D	*':' symbol* ':' symbol out of place in type definition string.
MVE_TYPEDEF_SYM_RBRACKET	0x672E	*']' symbol* ']' symbol out of place in type definition string.
MVE_TYPEDEF_SYM_LBRACKET	0x672F	*'[' symbol* '[' symbol out of place in type definition string.
MVE_TYPEDEF_NUM_ELTS	0x6730	*Empty Array* Array must have at least one element.
MVE_TYPEDEF_SYM_DIGIT	0x6731	*Numeric Digit Misplaced* Numeric digit is out of place in type definition string.
MVE_TYPEDEF_SYM_OTHER	0x6732	*Non-terminal Symbol Not in String* Invalid symbol in type definition string.
Other error codes:		
MVE_DATA_SPACE	0x6733	*User Supplied Data Space* Virtual Machine Upload has run out of user supplied data space in which to place destination.
MVE_VM_REQ_ERR	0x6734	*Could Not Send Request* Virtual Machine cannot send either an Upload or File (FileCopy or ObtainFile) request.
MVE_VMD_NOT_EMPTY	0x6735	*VMD Not Empty* Attempted to delete a VMD that is not empty of MMS objects.
MVE_BAD_APP_REF_FORM	0x6736	*App Ref Not Form 2* Using an unsupported Application Reference. This must be form 2 — MMS Object Identifier.
MVE_ASN1_TO_RT	0x6737	*ASN.1 to Runtime* ASN.1 to runtime decode failed.
MVE_ASN1_ENCODE_ERR	0x6738	*ASN1 Encode Func returned an error* ASN.1 encoding function returned an error.
MVE_ASN1_DECODE_ERR	0x6739	*ASN1 Decode Func returned an error* ASN.1 decoding function returned an error.
MVE_FP_SAVE_REQ_ERR	0x673A	*Error Saving Last Request* On a FileCopy unable to save last request.
MVE_FP_REQ_DONE_ERR	0x673B	*Error Marking Request as Returned* On a FileCopy unable to mark request as returned.
MVE_ADD_APPREF_ERR	0x673D	*APP_REF Error* Error adding an **APP_REF** structure to the list. See **Volume 3 — Module 10** for more information.
MVE_RESP_NOT_SUP	0x673E	*Response Not Supported* Specific response is not supported. Check **mmsop_en.h** to make sure specific service response is enabled.
MVE_REQ_NOT_SUP	0x673F	*Request Not Supported* Specific request is not supported. Check **mmsop_en.h** to make sure specific service request is enabled.
MVE_INVALID_PDU	0x6740	*Invalid PDU Construct* PDU sent is improperly formed.

DEFINED CONSTANT	HEX VALUE	MEANING
MVE_DOM_DISCARDED	0x6741	*Domain Discarded* Domain download was unsuccessful — domain segments are discarded.
MVE_DOM_PROTECTED	0x6742	*Domain is Protected by Server* Cannot delete indicated domain — protected by the server — download not done.
MVE_DOM_DEL_ERR	0x6743	*Server will not Kill Domain* Domain cannot be deleted by the server — download not done.
MVE_DEL_PI_ERR	0x6744	*Server will not Kill and Delete PI* An attempt to delete Program Invocations at the Server has caused the Server to recreate them — download not done.
MVE_RTAA_TYPE	0x6745	*Problem with RTAA type* Error creating Alternate Access runtime type.
MVE_RTAA_SIZE	0x6746	*m_max_rt_aa_ctrl Too Small* Too many elements in the Alternate Access runtime type.
MVE_AA_SELECT	0x6747	*Problem with AA Select* Cannot find the start element in a nested Alternate Access runtime type.
MVE_INVALID_ADL	0x6748	*ADL is invalid* Defined ADL is not correct.
MVE_DATA_TO_RT	0x6749	*ASN.1 Data to Runtime Conversion Problem* Error converting ASN.1 data to a runtime type.
PI_NAME	0x6770	*Invalid Program Invocation Name* A Program Invocation Object by that name already exists or a Program Invocation Object should exist but it doesn't.
PI_STATE	0x6771	*Invalid Program Invocation State* A bad PI state is returned for a given operation.
PI_NOT_REUSABLE	0x6772	*Program Invocation cannot be re-used* Cannot reset this Program Invocation.
PI_NOT_DELETABLE	0x6773	*Program Invocation cannot be deleted* Cannot delete this Program Invocation.
ME_PI_PROTECTION	0x6774	*Program Invocation Protection Error* Privilege is invalid to manage this Program Invocation.
ME_PI_INVALID_ID	0x6775	*Program Invocation has Invalid ID* Invalid MMS Identifier.

The following error codes pertain to exception conditions passed in the **errval** input to the **u_mmsexcept_ind** function. These types of errors are very serious. They usually require that the association be aborted over which the exception condition occurred in order to continue normal communications on the remainder of the channels and sometimes all communication must be stopped and MMS-EASE restarted.

DEFINED CONSTANT	HEX VALUE	MEANING
ERROR CODES FOR MMS EXCEPTIONS: E_MMS_EXCPT 0x6800		
MX_REJECT	0x6801	*Could not send REJECT* A protocol error was detected but MMS-EASE could not send a Reject PDU.
MX_BOARD_ERR	0x6802	*Board driver exception* MMS-EASE indicated to the user that a message was sent but the OSI interface board later rejected the message before making it to the network.
MX_LLP_EVENT_ERROR	0x6803	*Unknown LLP Event Code* An invalid event occurred at the LLP level.
MX_MEMORY_ALLOC	0x6804	*Memory Allocation* There was a memory allocation error that occurred while the virtual machine was attempting to allocate memory for a data structure.
MX_RUNTIME_TDEF	0x6805	*Runtime Type* There was a runtime type definition error detected by the virtual machine. This may result from an invalid variable type definition or if the application program overwrites the run time type definition by mistake.
MX_LLP_QUE_OVERFLOW	0x6806	*LLP Event Que* There was a queue overflow at the LLP level.
MX_MMSIND_QUE	0x6807	*MMS Indication Control Que* The **MMSREQ_IND** queue has overflowed.
MX_MMSCONF_QUE	0x6808	*MMS Confirm Control Que* The **MMSREQ_PEND** queue has overflowed.
MX_INTERNAL_ERROR	0x6809	*Internal MMS-EASE* There was an internal MMS-EASE error that cannot be classified in any other error code. This is usually fatal and requires restarting MMS-EASE.
MX_LLP_ADD_INFO_ALLOC	0x680A	*LLC Allocation Error* LLC Error — Unable to allocate address for remote node.

DEFINED CONSTANT	HEX VALUE	MEANING
The following error codes are passed to the **errval** parameter of the **u_llp_error_ind** function:		
ERROR CODES FOR ACSE LLP: E_LLP_ACSE 0x6900		
ACSE_RECEIVE_ERR	0x6901L	*Receive Error* Error receiving data over an ACSE channel.
ACSE_LISTEN_ERR	0x6902L	*Listen Error* Error trying to post a listen to an ACSE channel.
ACSE_DISCONNECT_RCV_ERR	0x6903L	*Receive Disconnect Error* Error attempting to disconnect over ACSE
ACSE_STOP_LISTEN_ERR	0x6904L	*Stop Listen Error* Error trying to stop a listen on an ACSE channel.
ACSE_ASS_REQ_ERR	0x6905L	*Associate Request Error* Error sending an ACSE associate request.
ACSE_ASS_RESP_ERR	0x6906L	*Associate Response Error* Error sending an ACSE associate response.
ACSE_REL_RESP_ERR	0x6907L	*Release Response Error* Error sending an ACSE release response.
ACSE_REL_REQ_ERR	0x6908L	*Release Request Error* Error sending an ACSE release request.
ACSE_SEND_REQ_ERR	0x6909L	*Send Request Error* Error sending data transfer request over ACSE.
ACSE_SET_DEBUG_ERR	0x690AL	*Debug Level Set Error* Error setting the debug level for ACSE.
ACSE_ABORT_ERR	0x690BL	*Abort* Error Error trying to abort an ACSE Channel.
ERROR CODES FOR LLP LLC: E_LLP_LLC 0x6B00L		
LLC_LISTEN_ERR	0x6B01L	*LLC Listen Error* Error posting a listen on a LLC Channel.
ERROR CODES FOR MMS DIRECTORY SERVICES: E_MMS_DS 0x6B00		
MDS_ERROR_INIT	0x6B01	*Error Initializing Directory Services* Directory Services cannot be initialized.
MDS_ERROR_INVALID_OPCODE	0x6B02	*Invalid Opcode* An invalid opcode passed to **mds_request**.
MDS_ERROR_INVALID_HANDLE	0x6B03	*Invalid Bind Handle* A bind handle passed to **mds_request** cannot be matched to a DSA.
MDS_ERROR_BAD_DN	0x6B04	*Invalid Distinguished Name* Distinguished Name format is incorrect
MDS_ERROR_RESULT	0x6B06	*Directory operation failed* A directory operation such as Bind or Read failed.
MDS_ERROR_BIND	0x6B07	*Error in Binding* A bind operation failed.
MDS_ERROR_UNBIND	0x6B08	*Error in Unbinding* An unbind operation failed.
MDS_ERROR_READ	0x6B09	*Error in Read* Read operation failed.

DEFINED CONSTANT	HEX VALUE	MEANING
MDS_ERROR_ATTR_TYPE_NAME	0x6B0A	*Invalid Attribute Type Name* Attribute Type Name passed to the **mds_request** is not recognized.
MDS_ERROR_DECODE	0x6B0B	*Decode Error* Error decoding Read result.
MDS_ERROR_BAD_PADDR	0x6B0C	*Invalid P-Address* The format of the Presentation Address returned in Read result is incorrect.

ACSE SUIC Error Codes

The following section describes the various error codes that can occur while using ACSE SUIC. These are the error codes returned by the driver, and SUIC itself. These errors are displayed in the following hexadecimal representations:

DEFINED CONSTANT	HEX VALUE	MEANING
ERROR CODES FOR ACSE SUIC: E_ACSESUIC 0xB00		
SE_CPU_ALLOC	0xB01	*System Memory Allocation Error* There is an error in allocating system memory.
SE_CHAN_ERR	0xB02	*Invalid Channel Number* Channel number assigned is invalid or not properly configured.
SE_QUE_FULL	0xB03	*Transmit Queue Full* SUIC send (Session TSDU) queue full.
SE_CHAN_CTRL	0xB04	*Invalid Channel State* There is a channel control error. Channel state is not OK for the operation. Check channel state for proper state (e.g., must be IDLE for Initiate).
SE_CTRL_ALLOC	0xB05	*Control Block Allocation Error* There is an error allocating control blocks, possibly out of control blocks.
SE_INSFRES	0xB06	*Insufficient Resources* There are insufficient resources allocated to the board.
SE_NAMENOTFOUND	0xB07	*AR NAME not found in local DIB* Please check DIB file or utility to make sure that the referenced AR Name is properly configured.
SE_CHANNOTREG	0xB08	*Channel not Registered* Invalid channel state — please check code or demo configuration file to make sure that the channel is properly registered.
SE_NOPARAM	0xB09	*Mandatory Parameter Missing* Make sure all the necessary parameters are negotiated for this channel such as AP Context.
SE_INVPARAM	0xB0A	*Invalid Parameter Supplied* A negotiated parameter for this channel is not valid or supported.
SE_INVPDU	0xB0B	*Invalid PDU* Invalid PDU encountered when decoding.

DEFINED CONSTANT	HEX VALUE	MEANING
SE_INVASN	0xB0C	*Invalid Abstract Syntax Name* Abstract Syntax Name is not specified, or is invalid.
SE_INVTSN	0xB0D	*Invalid Transfer Syntax Name* Transfer Syntax Name is not specified, or is invalid.
SE_INVCTXT	0xB0E	*Invalid AP Context Name* The AP Context Name is not specified, or is invalid.
SE_INVPCI	0xB0F	*Invalid PCI* Presentation Context Identifier is not specified, or is invalid.
SE_INVOBJID	0xB10	*Invalid OBJECT IDENTIFIER* The object identifier is not specified, or is invalid. Check DIB file.
SE_SIMPLY_ENC	0xB11	*Simply Encoded Data* Invalid ASN.1 data.
SE_MSGSIZETOOBIG	0xB12	*User-defined **s_msgsize** is Larger than Max. Possible* Defined maximum message is larger than allowed by the stack.
SE_NOLLP	0xB13	*LLP is not running* Lower Layer Provider is not running
SE_ACTIVATE	0xB1C	*Name Activation problem* Problem activating an AR Name (standard SUIC error).
SE_DEACTIVATE	0xB1D	*Name deactivation problem* Problem deactivating an AR Name (standard SUIC error).
The following error codes are use with Directory Services (x.500 and LDAP):		
SE_NOTBOUND	0xB14	*Not bound to a DSA* A call to either **mds_set_def_userid** or **ldap_set_def_userid** failed.
SE_MDS_UNEXPECTED	0xB15	*Unexpected MDS result* A read operation returned an unexpected number of results.
SE_MDS_INVALID_SYNTAX	0xB16	*Invalid attribute syntax* Attributes set in the DUA's Directory Schema as defined in the file **xds.ini** are invalid.

DEFINED CONSTANT	HEX VALUE	MEANING
The following error codes are used with the MMS-EASE Security Toolkit		
SE_BAD_MECH_NAME	0xB17	*Unrecognized authentication mechanism name* The Remote AR Name is configured to support an unrecognized Authentication Mechanism Name.
SE_SECURITY CONTEXT	0xB18	*Security context could not be created* A call to **create_calling_context** failed.
SE_SECURE_DATABASE	0xB19	*Security database access error* Error accessing the Security database or the data stored in the database is inconsistent.
SE_ENCRYPT	0xB1A	*DES CBC Encryption problem* A call to **descbc_encrypt** failed.
SE_DECRYPT	0xB1B	*DES CBC Decryption problem* A call to **descbc_decrypt** failed.
ERROR CODES FOR ACSE SUIC (Implementation Specific): E_ACSESUIC_IS 0x1500		
The following error codes are Marben-specific — any MMS-EASE-132-xxx will use them.		
SE_SHMALLOC	0x1501	*Shared Memory Allocation Error* Error in request to allocate shared memory.
SE_BADPROTO	0x1504	*Inconsistent Protocols* An attempt is made to establish a connection between an OSI AR Name and a TCP/IP AR Name.
SE_SERVICENOTINSTALLED	0x1505	*Required Service Not Installed* OSILLC Driver and/or TCP/IP Stack not installed but OSI and/or TCP/IP are configured as required.
SE_MARBENPORT_ERR	0x1506	*The Stack reported a Request* An error has been reported by the lower layers of the stack. See the log file (OSILL2D.LOG) for more information.
ERROR CODES FOR ACSE SUIC EXCEPTIONS: EX_ACSESUIC 0xF00		
SX_INTERNAL	0xF01	*Internal Error* User error detected by provider.
SX_CPU_ALLOC	0xF02	*System Memory Allocation Error* There is an exception in allocating system memory.
SX_CTRL_ALLOC	0xF03	*Control Block Allocation Error* There is an exception allocating control blocks, possibly out of control blocks.
SX_NOCTRL	0xF04	*Out of Control Blocks* There is an exception indicating out of control blocks.
SX_NODATA	0xF05	*Out of Data Buffers* There is an exception indicating out of data buffers.
SX_QUE_FULL	0xF06	*Transmit Queue Full* SUIC send queue is full. This usually occurs when large amounts of InformationReports are sent.
SX_INVPDU	0xF07	*Invalid PDU* Sending abort due to invalid PDU data field.
SX_INVCHAN	0xF08	*Invalid Channel Number* Channel number assigned is invalid

DEFINED CONSTANT	HEX VALUE	MEANING
SX_BADCB	0xF09	*Unknown Control Block* Control Block is not valid.
SX_ENCERR	0xF0A	*Encode Error* ASN.1 error in encoding.
SX_CTRL_FREE	0xF0B	*Control Block Free Error* Exception in freeing control block.
SX_DATA_FREE	0xF0C	*Data Buffer Free Error* Exception in freeing data block.
SX_DATABUF_OVRFLW	0xF0D	*Exceeds s_msgsize* Data being received exceeds **s_msgsize** — the defined maximum message is larger than allowed by the stack.

ERROR CODES FOR ACSE2: E_ACSE2 0x3000

The following error codes are returned from the Lean-T or Reduced stack ACSE layer.

E_ACSE_ENC_ERR	0x3001	*ACSE Encode Error* Error in ASN.1 encoding.
E_ACSE_SEND_ERR	0x3002	*ACSE Send Error* Error in sending ACSE.
E_ACSE_INVALID_CONN_ID	0x3003	*Invalid Connection ID* Connection ID is not valid.
E_ACSE_INVALID_STATE	0x3004	*Invalid State* ACSE is not in valid state.
E_ACSE_INVALID_PARAM	0x3005	*Invalid Parameter* Parameter sent is invalid.
E_ACSE_BUFFER_OVERFLOW	0x3006	*Buffer Overflow Error* ASN.1 buffer overflow.
E_ACSE_MEMORY_ALLOC	0x3007	*Error Allocating Memory* Memory allocation failed.

EXCEPTION CODES FOR ACSE2: EX_ACSE2

EX_ACSE_DECODE	0x3081	*ACSE Decode Error* Error in ASN.1 decode.
EX_ACSE_INVALID_STATE	0x3082	*Invalid State* ACSE is not in valid state.

Lower Layer Error Codes

The following section describes the various error codes that can occur at the Lower Layers of the stack. These errors are displayed in the following hexadecimal representations:

DEFINED CONSTANT	HEX VALUE	MEANING
ERROR CODES FOR TP4 (Transport): E_TP4 0x1200		
SE_CPU_ALLOC	0x1201	*System Memory Allocation Error* There is an error in allocating system memory.
SE_CHAN_ERR	0xB02	*Invalid Channel Number* Channel number assigned is invalid or not properly configured.
SE_QUE_FULL	0xB03	*Transmit Queue Full* SUIC send (Session TSDU) queue full.
SE_CHAN_CTRL	0xB04	*Invalid Channel State* There is a channel control error. Channel state is not OK for the operation. Check channel state for proper state (e.g., must be IDLE for Initiate).
SE_CTRL_ALLOC	0xB05	*Control Block Allocation Error* There is an error allocating control blocks, possibly out of control blocks.
SE_INSFRES	0xB06	*Insufficient Resources* There are insufficient resources allocated to the board.
SE_NAMENOTFOUND	0xB07	*AR NAME not found in local DIB* Please check DIB file or utility to make sure that the referenced AR Name is properly configured.
SE_CHANNOTREG	0xB08	*Channel not Registered* Invalid channel state — please check code or demo configuration file to make sure that the channel is properly registered.
SE_NOPARAM	0xB09	*Mandatory Parameter Missing* Make sure all the necessary parameters are negotiated for this channel such as AP Context.
SE_INVPARAM	0xB0A	*Invalid Parameter Supplied* A negotiated parameter for this channel is not valid or supported.
SE_INVPDU	0xB0B	*Invalid PDU* Invalid PDU encountered when decoding.
SE_INVASN	0xB0C	*Invalid Abstract Syntax Name* Abstract Syntax Name is not specified, or is invalid.
SE_INVTSN	0xB0D	*Invalid Transfer Syntax Name* Transfer Syntax Name is not specified, or is invalid.
SE_INVCTXT	0xB0E	*Invalid AP Context Name* The AP Context Name is not specified, or is invalid.
SE_INVPCI	0xB0F	*Invalid PCI* Presentation Context Identifier is not specified, or is invalid.

DEFINED CONSTANT	HEX VALUE	MEANING
SE_INVOBJID	0xB10	*Invalid Object Identifier* The object identifier is not specified, or is invalid. Check DIB file.
SE_SIMPLY_ENC	0xB11	*Simply Encoded Data* Invalid ASN.1 data.
SE_MSGSIZETOOBIG	0xB12	*User-defined s_msgsize Larger than Max. Possible* Defined maximum message is larger than allowed by the stack.
SE_NOLLP	0xB13	*LLP Not Running* Lower Layer Provider is not running
SE_ACTIVATE	0xB1C	*Name Activation Problem* Problem activating an AR Name (standard SUIC error).
SE_DEACTIVATE	0xB1D	*Name Deactivation Problem* Problem deactivating an AR Name (standard SUIC error).
The following error codes are use with Directory Services (x.500 and LDAP):		
SE_NOTBOUND	0xB14	*Not Bound To A DSA* A call to either **mds_set_def_userid** or **ldap_set_def_userid** failed.
SE_MDS_UNEXPECTED	0xB15	*Unexpected MDS Result* A read operation returned an unexpected number of results.
SE_MDS_INVALID_SYNTAX	0xB16	*Invalid Attribute Syntax* Attributes set in the DUA's Directory Schema as defined in the file **xds.ini** are invalid.
The following error codes are used with the MMS-EASE Security Toolkit		
SE_BAD_MECH_NAME	0xB17	*Unrecognized Authentication Mechanism Name* The Remote AR Name is configured to support an unrecognized Authentication Mechanism Name.
SE_SECURITY CONTEXT	0xB18	*Security Context Could Not Be Created* A call to **create_calling_context** failed.
SE_SECURE_DATABASE	0xB19	*Security Database Access Error* Error accessing the Security database or the data stored in the database is inconsistent.
SE_ENCRYPT	0xB1A	*DES CBC Encryption Problem* A call to **descbc_encrypt** failed.
SE_DECRYPT	0xB1B	*DES CBC Decryption Problem* A call to **descbc_decrypt** failed.

DEFINED CONSTANT	HEX VALUE	MEANING
ERROR CODES FOR Connection-Oriented Session Protocol (COSP): E_COSP	0x3200	
COSP_ERR_BIND_STATE	0x3201	*Invalid Bind State* Transport layer is already bound.
COSP_ERR_TP4_RET	0x3202	*TP4 Error* Transport layer returned an error in bind or unbind operation.
COSP_ERR_INV_TP4_ADDR	0x3203	*Invalid Transport Address* Length of Transport Address is invalid in bind operation.
COSP_ERR_INV_CON_STATE	0x3204	*Invalid Connection State* Connection State is not valid.
COSP_ERR_INV_SSEL	0x3205	*Invalid Local SSEL Length* Local Session Selector length is invalid.
COSP_ERR_INV_UDATA_LEN	0x3206	*Invalid User Data Length* Error in length of user data in negative response.
COSP_ERR_INV_POINTER	0x3207	*Invalid Pointer to Encode Buffer* Pointer to encode buffer is invalid or null.
The following are COSP PDU decode errors:		
COSP_ERR_DEC_INV_SPDU	0x3210	*Invalid Session PDU Received* Length of Session PDU is not supported.
COSP_ERR_DEC_INV_LEN	0x3211	*Invalid Session PDU Length* Decoded length does not match indicated length of Session PDU.
COSP_ERR_DEC_INV_PI_CODE	0x3212	*Invalid Parameter Code* Received parameter code in Session PDU is not supported.
COSP_ERR_DEC_INV_LOC_SSEL	0x3213	*Invalid SSEL Received* Received Session Selector is not supported.
COSP_ERR_DEC_INV_PROT_OPT	0x3214	*Invalid Protocol Option* Extended concatenation is not supported in selected protocol option.
COSP_ERR_DEC_INV_SEG	0x3215	*Invalid Segmentation Option* Segmenting of Session Data Units not supported.
COSP_ERR_DEC_INV_PROT_VER	0x3216	*Invalid Protocol Version* Currently Version 2 of ISO 8473 is supported.
COSP_ERR_DEC_INV_FUN_UNITS	0x3217	*Invalid Session User Requirement* Duplex functional unit must be proposed.
COSP_ERR_DEC_INV_RF_UDATA	0x3218	*Invalid User Data* Error when refuse reason is not equal to 2 — protocol error.
COSP_ERR_DEC_INV_AB_RP	0x3219	*Invalid Protocol Error* Reflect parameter is equal to 0 when Abort SPDU reason is set to protocol error.

DEFINED CONSTANT	HEX VALUE	MEANING
ERROR CODES FOR Connection-Less Network Protocol (CLNP): E_CLNP	0x3400	
CLNP_ERR_CFG_FILE	0x3400	*Configuration File Errors* Required parameters not configured — local MAC address and local NSAP.
CLNP_ERR_NOT_INIT	0x3401	*CLNP Not initialized* Protocol not started.
CLNP_ERR_MEM_ALLOC	0x3402	*Error in Memory Allocation* Cannot allocate memory.
CLNP_ERR_NULL_PTR	0x3403	*Null Pointer Error* Null Pointer passed to a **clnp**_ function.
CLNP_ERR_NSAP_LEN	0x3404	*NSAP Length Error* NSAP length is 0 or more than allowed value. This is an unreoverable error during CLNP initialization.
CLNP_ERR_LIFETIME	0x3405	*Invalid PDU Lifetime* Recoverable error during CLNP initialization. Lifetime value will be set to default.
CLNP_ERR_LIFETIME_DEC	0x3406	*Invalid PDU Lifetime Decrement* Recoverable error during CLNP initialization. Lifetime decrement value will be set to default.
CLNP_ERR_ESH_CFG_TIMER	0x3407	*Invalid ESH Configuration Timer* Recoverable error during CLNP initialization. End System Holder timer will be set to default value.
CLNP_ERR_ESH_DELAY	0x3408	*Invalid Delay Time for First ESH* Recoverable error during CLNP initialization. Delay time will be set to default value.
CLNP_ERR_MAC_ADDR	0x3409	*Local MAC Address Not Configured.* Must have a local MAC Address — this is required for ADLC sub-network. Unrecoverable error during CLNP initialization.
CLNP_ERR_UDATA_LEN	0x3410	*CNLP-User Data Length Too Large* User data exceeds set maximum length.
The following errors are specific to CLNP PDU parsing (decode):		
CLNP_ERR_PDU_MAC_ADDR	0x3420	*Error Decoding MAC Address PDU* The NPDU MAC Address is not a local MAC Address or cannot decode ALL End Systems Addresses.
CLNP_ERR_PDU_ID	0x3421	*Invalid PDU ID* Not a supported PDU. Currently ISO 8473 and ISO 9542 standards are supported.
CLNP_ERR_PDU_VER	0x3422	*Invalid PDU Version* Not a supported PDU version. Currently ISO 8473 and ISO 9542 standards are supported.
CLNP_ERR_PDU_TYPE	0x3423	*Invalid PDU Type* Not a supported PDU type. Currently DT, ER, ESH, and ISH PDUs are supported.
CLNP_ERR_PDU_LEN	0x3424	*Invalid PDU Length* Received PDU length does not match the length indicated by the sub-network.

DEFINED CONSTANT	HEX VALUE	MEANING
CLNP_ERR_PDU_EXPIRED	0x3425	*PDU Expired* DT (Data Type) or ER (Error) PDU's lifetime has expired.
CLNP_ERR_PDU_NSAP_ADDR	0x3426	*Error NSAP Addressing To PDU* PDU is improperly addressed to a NSAP that is not assigned locally.
CLNP_ERR_PDU_SEGMENTING	0x3427	*Error Segmenting PDUs* Segmented PDUs are not supported — PDUs must arrive in one packet.
CLNP_ERR_PDU_CHECKSUM	0x3428	*Error PDU Checksum* PDU checksum verification failed.
CLNP_ERR_PDU_LAST_SEG	0x3429	*Segmented PDU Error* Last segment bit not set — indicating an unsupported segmented PDU.
CLNP_ERR_PDU_ER_PDU	0x342A	*Error ER PDU* Code not compiled for ER (Error) PDU processing.
The following errors are specific to LLC Encode/Decode:		
LLC_ERR_SRC_ADDR	0x3481	*Error LLC Source Address* LLC Source address is invalid.
LLC_ERR_DEST_ADDR	0x3482	*Error LLC Destination Address* LLC Destination address is invalid.
LLC_ERR_CONTROL	0x3483	*Error LLC Header Control Field* LLC Header Control field is invalid.
ERROR CODES FOR SUB-NETWORK INTERFACE: E_SUBNET	0x3500	
SNET_ERR_INIT	0x3501	*Error Initializing Sub-Network Interface* Sub-network interface not available.
SNET_ERR_WRITE	0x3502	*Sub-Network Write Function Failed* Cannot write to Sub-network.
SNET_ERR_READ	0x3503	*Sub-Network Read Function Failed* Cannot read from Sub-network.
SNET_ERR_MAC_INVALID	0x3504	*Invalid MAC Address* Unable to obtain All End Systems, All Intermediate Systems or local MAC Address.
SNET_ERR_FRAME_LEN	0x3505	*Frame Length Error* Received more data than can be reserved for the buffer.
SNET_ERR_UDATA_LEN	0x3506	*User Data Length Error* Invalid length of send data — length of data is invalid.

DEFINED CONSTANT	HEX VALUE	MEANING
The following Sub-network errors are specific to the Ethernet driver:		
SNET_ERR_DRV_OPEN	0x3520	*Open Driver Command Failed* Cannot install the **osillc.vxd** (Win95/NT) or **OSILLC$** (Win 3.x) driver.
SNET_ERR_DRV_LOC_MAC	0x3521	*Driver Error for Local MAC Address* Failure to obtain local MAC address from the ethernet board.
SNET_ERR_DRV_ADD_ES_ADDR	0x3522	*ES Address Driver Error* Failure to activate All End System Address.
SNET_ERR_DRV_BIND_LSAP	0x3523	*Failure to bind to LSAP* Cannot bind **OSILLC$** (Win 3.x) driver to LSAP.
SNET_ERR_DRV_POST_BUFS	0x3524	*Failure to Post Buffers to Driver* **OSILLC$** (Win 3.x) driver cannot post buffers.
ERROR CODES FOR ADLC: E_ADLC 0x3600		
E_ADLC_INVALID_DEV_NAME	0x3601	*Invalid Device Name* Device name passed in function parameter does not exist.
E_ADLC_INVALID_DEV_ADDR	0x3602	*Invalid Device Address* Device address passed in function parameter does not exist.
E_ADLC_DEVICE_EXISTS	0x3603	*Device Already Exists* Request to add a device failed because device already exists.
E_ADLC_DEV_DOES_NOT_EXIST	0x3604	*Device Does Not Exist* Request to remove a device failed because device does not exist.
E_ADLC_INVALID_CONN_PAIR	0x3606	*Invalid Connection Pair* Request for connection status failed.
E_ADLC_CON_EXISTS	0x3607	*ADLC Connection Already Exists* Request for ADLC connection failed because connection already exists.
E_ADLC_CON_DOES_NOT_EXIST	0x3608	*ADLC Connection Does Not Exist* Request for ADLC connection failed because no connection exists.
E_ADLC_TX_BUF_FULL	0x3609	*Transmission Buffer Full* Cannot process a write request for this connection.
E_ADLC_CANT_POLL_FASTER	0x360A	*Cannot Poll Faster* Request to change poll rate failed because it is being polled at the highest rate.
E_ADLC_CANT_POLL_SLOWER	0x360B	*Cannot Poll Slower* Request to change poll rate failed because it is being polled at the slowest rate.
E_ADLC_INVALID_POLL_DELTA	0x360C	*Invalid Parameter in Change Poll Rate* Check the parameter in the request function to change the poll rate.
E_ADLC_INTERNAL_ERR	0x360D	*ADLC Internal Error* Serious programming error. Contact SISCO Technical Support.

DEFINED CONSTANT	HEX VALUE	MEANING
E_ADLC_DEVICE_NOT_POLLED	0x360E	*Device Not Polled* Any request regarding an inquiry or change in poll rate will fail.
E_ADLC_INVALID_HQUEUE	0x360F	*Invalid Queue Handle* Another process using ICP has passed an invalid queue handle.
E_ADLC_INVALID_PORT	0x3610	*Invalid Port* Port number is either not configured or system returned an error while initializing or terminating the port.
E_ADLC_INVALID_POLL_FREQ	0x3611	*Invalid Poll Frequency* Poll frequency passed in parameter for adding device is not valid.
E_ADLC_RESPONSE_TIMEOUT	0x3614	*Response Timeout* Remote node timed out while waiting for a response.
E_ADLC_INVALID_CMD	0x3615	*Invalid Command* Command code sent by the user is invalid.
E_ADLC_CANT_INIT_PORT	0x3616	*Cannot Initialize Port* The operating system returned an error when attempting to initialize the port.
E_ADLC_CREATE_THREAD	0x3617	*Cannot Create Thread* Operating system could not create thread .
E_ADLC_CANT_INIT_TAPI	0x3618	*Cannot Initialize TAPI.* Operating system could not initialize Telephone API.
E_ADLC_TEL_LINE_UNAVAIL	0x3619	*Telephone Line Unavailable* Line is busy or defective.
E_ADLC_CANT_INIT_TEL_LINE	0x361A	*Cannot Initialize Telephone Line* Operating system is unable to initialize the telephone line.
E_ADLC_PORT_EXISTS	0x361B	*ADLC Port Already Exists* Request to add port failed because port already exists.
E_ADLC_PORT_DOES_NOT_EXIST	0x361C	*ADLC Port Does Not Exist* Request to remove port failed because port does not exist.

DEFINED CONSTANT	HEX VALUE	MEANING
ERROR CODES FOR RLI_IPC: 0x3700		
The following error codes are returned from the Radio Link Interface IPC:		
ERR_IPC_INVALID_TASK_ID	0x3701	*Invalid Task ID* See **rli_ipc.h** file for the list of valid task ids.
ERR_IPC_CREATE_QUEUE	0x3702	*Error Creating IPC Resource Queue* Possible reasons: queue name too long, queue already exists, too many queues created. Normally the IPC task creates at least on queue for IPC messages.
ERR_IPC_CREATE_SEM	0x3703	*Error Creating Semaphore* System function to create an event failed.
ERR_IPC_PEEK_QUE	0x3704	*Attempt to look at Queue Failed* Check for messages in queue has failed — usually due to invalid queue handle.
ERR_IPC_MAX_IN_MUXWAIT	0x3705	*Max Number of ICP Resources Exceeded* Array of created IPC resources (semaphores, timers, queues) has been exhausted. Default is set by **IPC_MAX_SEM_CNT** (default = 64).
ERR_IPC_ADD_TO_MUXWAIT	0x3706	*Attempt to add to IPC Resources Failed* Cannot add to the array of created IPC resources.
ERR_IPC_NO_DEF_QUENAME	0x3707	*Error No Default Queue Name* No default queue name found for the specific RLI Task ID. See **rli_ipc.h** file for a list of valid functions used in queue management.
ERR_IPC_INVALID_EVTYPE	0x3708	*Invalid Event Type* Possible corruption of the IPC resource array.
ERR_IPC_WAIT_MUXWAIT	0x3709	*Error Waiting on IPC Resources* Call to **ipcWaitMuxSemEvent** failed. The system function waiting for multiple semaphores failed.
ERR_IPC_INVALID_IDX	0x370A	*Invalid Index* Invalid handle in either **ipcStartTimer** or **ipcEndTimer** functions or serious problem in the IPC library function execution.
ERR_IPC_RESET_SEM	0x370B	*Error Resetting Semaphore* Call to system function to reset semaphore has failed.
ERR_IPC_READ_QUE	0x370C	*Error Reading Queue* Invalid queue handle.
ERR_IPC_GET_SHARED_MEM	0x370D	*Error Getting Shared Memory* OS/2 specific — cannot get access to shared memory.
ERR_IPC_MAX_IN_OPEN_RES	0x370E	*Maximum Open Resources Exceeded* The array of open IPC resources (semaphores, timers, queues) is full. Exceeded **IPC_MAX_OPEN_RES** (default = 128).
ERR_IPC_OPEN_QUEUE	0x370F	*Error Opening Queue* Too many queues open. Queue has not been created by any process. System function to open event attached to queue has failed.
ERR_IPC_INVALID_RESTYPE	0x3710	*Invalid IPC Resources* Possible corruption of the array of IPC resources.

DEFINED CONSTANT	HEX VALUE	MEANING
ERR_IPC_ALLOC_SHARED_MEMORY	0x3711	*Error in Allocating Shared Memory* Cannot allocate shared memory.
ERR_IPC_GIVE_SHARED_MEMORY	0x3712	*Error in Accessing Shared Memory* Giviing access to shared memory to another task failed. Invalid pointer in **ipcGiveSMem** call.
ERR_IPC_PID_NOT_FOUND	0x3713	*Process ID Not Found* OS/2 specific — queue to a process must be opened before access to a shared memory is given.
ERR_IPC_WRITE_QUE	0x3714	*Error Writing to Queue* Invalid handle — IPC 32 driver error.
ERR_IPC_FREE_SHARED_MEMORY	0x3715	*Error Freeing Shared Memory* Null or invalid pointer to memory or program is corrupt.
ERR_IPC_QUEUE_NOT_OPEN	0x3716	*Queue Not Open* Cannot access queue — not previously opened.
ERR_IPC_START_TIMER	0x3717	*Start Timer Error* System function to set time event has failed.
ERR_IPC_TIMER_NOT_FOUND	0x3718	*Timer Not Found* This error is returned when checking for duplicate timer handles in the array of IPC resources. This may occur if timer interval created is too small (50 ms or less).
ERR_IPC_EVSEM_NOT_FOUND	0x3719	*Event Semaphore Not Found* Invalid handle to event semaphore.
ERR_IPC_QUEUE_NOT_FOUND	0x371B	*Queue Not Found* Cannot perform a close queue operation.
ERR_IPC_INVALID_HQUEUE	0x371C	*Invalid Queue Handler* Handle to queue is invalid.
ERR_IPC_INVALID_HSEM	0x371D	*Invalid Semaphore Handler* Handle to event semaphore is invalid.
ERR_IPC_POST_SEM_FAILED	0x371E	*Failure To Post Semaphore* Open or post event semaphore operation has failed.
ERR_IPC_ADD_EXITFUNC_FAILED	0x371F	*Exit Function Add Failed* Failure to Add Exit Function to the list of functions executed on exit.
ERR_IPC_GET_PROC_INFO	0x3721	*Error Getting Process Info* Get Process Priority Information failed.
ERR_IPC_SET_PROC_PRIORITY	0x3722	*Error Setting Process Priority* Setting Process Priority failed.
ERR_IPC_CREATE_SHARED_MEM	0x3723	*Error Creating Shared Memory* Cannot create shared memory.
ERR_IPC_ATTACH_SHARED_MEM	0x3724	*Error Attaching Shared Memory* Cannot attach shared memory.
ERR_IPC_DELETE_SHARED_MEM	0x3725	*Error Deleting Shared Memory* Cannot delete shared memory.

MMS Error Codes

The following error codes are MMS error codes that should be used when sending error responses to received indications. These error codes should be used for the **`mp_cancel_err`**, **`mp_conclude_err`**, **`mp_err_resp`**, and **`mp_init_err`** functions. These codes are defined by the MMS standard, taken from the ISO 9506 protocol.

CODE	MEANING	
ERROR CLASS = 0 VMD STATE PROBLEMS This error class is returned whenever the state of the VMD is such that the requested service may not be executed.		
0	OTHER	Returned due to a reason other than any of those identified for this error class.
1	VMD-STATE-CONFLICT	Returned when a request is made that alters the VMD in a way that conflicts with the current state.
2	VMD-OPERATIONAL-PROBLEM	Returned when a request is made that may not be honored due to VMD operational problems.
3	DOMAIN-TRANSFER-PROBLEM	Returned when a transmitted Load Data contains an inconsistency. This may prevent it from being used.
4	STATE-MACHINE-ID-INVALID	Returned when there is no state machine associated with the state machine ID.
ERROR CLASS = 1 APPLICATION REFERENCE PROBLEMS This error class is returned with respect to associations other than those established between a MMS client and a MMS server.		
0	OTHER	Returned due to a reason other than any of those identified for this error class.
1	APPLICATION-UNREACHABLE	Returned when reference application is unreachable.
2	CONNECTION-LOST	Returned when connection to specific application is lost before service was finished.
3	APPLICATION-REFERENCE-INVALID	Returned due to invalid application reference.
4	CONTEXT-UNSUPPORTED	Returned when referenced application does not support indicated application context.

CODE	MEANING	
ERROR CLASS = 2 DEFINITION PROBLEMS		
This error class is returned when there are problems with Object definitions.		
0	OTHER	Returned due to a reason other than any of those identified for this error class.
1	OBJECT-UNDEFINED	Returned when object with stated name does not exist.
2	INVALID-ADDRESS	Returned when specified address is invalid because format is incorrect or out of range. Applies only to unnamed variable objects, and only when parameter VADR is chosen.
3	TYPE-UNSUPPORTED	Returned when unsupported or inappropriate type is specified.
4	TYPE-INCONSISTENT	Returned when specified type is inconsistent with service or referenced object.
5	OBJECT-EXISTS	Returned when defined object already exists.
6	OBJECT-ATTRIBUTE-INCONSISTENT	Returned when specified object has inconsistent attributes.
ERROR CLASS = 3 RESOURCE PROBLEMS		
This error class is returned in response to a service which requests the assignment of resources that are not available for assignment.		
0	OTHER	Returned due to a reason other than any of those identified for this error class.
1	MEMORY-UNAVAILABLE	Returned when memory resources such as tables used to maintain definitions of names, event actions, journals, files are not available.
2	PROCESSOR-RESOURCE-UNAVAILABLE	Returned when CPU resources used to support maintenance of states is not available.
3	MASS-STORAGE-UNAVAILABLE	Returned when storage for additional detail is lost.
4	CAPABILITY-UNAVAILABLE	Returned when one or more capabilities are insufficient.
5	CAPABILITY-UNKNOWN	Returned when one or more capabilities are unknown.
ERROR CLASS = 4 SERVICE PROBLEMS		
This error class is returned whenever there are problems with the service primitives.		
0	OTHER	Returned due to a reason other than any of those identified for this error class.
1	PRIMITIVES-OUT-OF-SEQUENCE	Returned when service primitive sequence is invalid.
2	OBJECT-STATE-CONFLICT	Returned when current object state does not permit a response for the specified service request.
3	PDU-SIZE (DIS only)	This value is reserved for future definition in IS specification.
4	CONTINUATION-INVALID	Returned when file name to continue after is not a member of the group of files included in the file specification.

CODE	MEANING	
5	OBJECT-CONSTRAINT-CONFLICT	Returned when current constraints on an object prevent execution of a service request.

ERROR CLASS = 5 SERVICE PREEMPT PROBLEMS
This error class is returned whenever a service is preempted due to a cancel request or due to some local reasons.

0	OTHER	Returned due to a reason other than any of those identified for this error class.
1	TIMEOUT	Returned when service is canceled due to user-defined timeout.
2	DEADLOCK	Returned when service is canceled by VMD to prevent a deadlock.
3	CANCEL	Returned when a service is canceled.

ERROR CLASS = 6 TIME RESOLUTION PROBLEMS
This error class is returned in response to a service that requests a time resolution that is not supportable by the responding MMS user.

0	OTHER	Returned due to a reason other than any of those identified for this error class.
1	INSUPPORTABLE-TIME-RESOLUTION	Returned for request for unsupportable time resolution.

ERROR CLASS = 7 ACCESS PROBLEMS
This error class is returned whenever the requested service to an object was incorrectly specified.

0	OTHER	Returned due to a reason other than any of those identified for this error class.
1	OBJECT-ACCESS-UNSUPPORTED	Returned when object is not defined for requested access.
2	OBJECT-NON-EXISTENT	Returned for non-existent object.
3	OBJECT-ACCESS-DENIED	Returned when MMS client has insufficient privilege to request the operation.
4	OBJECT-INVALIDATED	Returned when attempted access references defined object that has undefined reference attribute.

CODE	MEANING	
ERROR CLASS = 8 INITIATE PROBLEMS		
This error class is returned when problems are encountered with the Initiate service.		
0	OTHER	Returned due to a reason other than any of those identified for this error class.
1	VERSION-INCOMPATIBLE (DIS only)	This value is reserved for future definition in IS specification.
2	MAX-SEGMENT-INSUFFICIENT (DIS only)	This value is reserved for future definition in IS specification.
3	MAX-SERVICES-OUTSTANDING-CALLING -INSUFFICIENT	Returned when proposed max services calling parameter too small for (**max_pend_req**) communication.
4	MAX-SERVICES-OUTSTANDING-CALLED-INSUFFICIENT	Returned when proposed max services called parameter too small for (**max_pend_resp**) communication.
5	SERVICE-CBB-INSUFFICIENT	Returned when service CBB necessary for communication is missing from proposed list.
6	PARAMETER-CBB-INSUFFICIENT	Returned when parameter CBB necessary for communication is missing from proposed list.
7	NESTING-LEVEL-INSUFFICIENT	Returned when proposed Data Structure Nesting level too small for communication.
ERROR CLASS = 9 CONCLUDE PROBLEMS		
This error class is returned when problems are encountered with the Conclude service.		
0	OTHER	Returned due to a reason other than any of those identified for this error class.
1	FURTHER-COMMUNICATION-REQUIRED	Returned when there are currently confirmed service requests for which responses have not been generated or an Upload State Machine exists.
ERROR CLASS=10 CANCEL PROBLEMS		
This error class is returned when problems are encountered with the Cancel service.		
0	OTHER	Returned due to a reason other than any of those identified for this error class.
1	INVOKE-ID-UNKNOWN	Returned when there is no confirmed service request for which a protocol machine exists with the specified Invoke ID.
2	CANCEL-NOT-POSSIBLE	Returned when cancel cannot be performed according to service requirements.

CODE	MEANING	
ERROR CLASS=11 FILE PROBLEMS		
This error class is returned for errors resulting from operations with files.		
0	OTHER	Returned due to a reason other than any of those identified for this error class.
1	FILE-NAME-AMBIGUOUS	Returned when trying to access a file through its name and this name contains "wildcard" symbols and more than one filename exists that meets the criteria.
2	FILE-BUSY	Returned when file is busy.
3	FILE-NAME-SYNTAX-ERROR	Returned when file name is incorrect syntactically.
4	CONTENT-TYPE-INVALID	Returned when file is not unstructured binary.
5	POSITION-INVALID	Returned when initial position is past the end of the file.
6	FILE-ACCESS-DENIED	Returned when access to the requested file is denied.
7	FILE-NON-EXISTENT	Returned when attempt to access a file that does not exist.
8	DUPLICATE-FILENAME	Returned when attempting to create a file with the same name as one that already exists in the filestore.
9	INSUFFICIENT-SPACE-IN-FILESTORE	Returned when attempting to add a file to the filestore, and there is no additional space to accommodate this file.
ERROR CLASS = 12 OTHER PROBLEMS		
This error class is returned for device-specific errors. The error codes for this class of errors are for future extensions and errors not addresses by any existing class.		

The following codes are the possible service specific errors that can occur for those services that require additional error information. These codes are defined by the MMS standards and would be used in the **ss_error_val** member of the **adtnl_err_info** structure. See **Volume 3 — Module 11 — Error Handling** for more information.

CODE	MEANING
OBTAINFILE ERROR (0)	
0	Source file problem
1	Destination file problem
START ERROR (1), or STOP ERROR (2), or RESUME ERROR (3), or RESET ERROR (4)	
	For these errors you should report the current state of the program invocation as shown.
0	Non existent
1	Unrunnable
2	Idle
3	Running
4	Stopped
5	Starting
6	Stopping
7	Resuming
8	Resetting
DELETE VARIABLE ACCESS ERROR (5)	
	Use the number of variables actually deleted
DELETE NAMED VARIABLE LIST ERROR (6)	
	Use the number of named variable lists actually deleted
DELETE NAMED TYPE ERROR (7)	
	Use the number of named types actually deleted
FILE RENAME PROBLEM (8)	
0	Source file problem
1	Destination file problem

MMS Reject Codes

The following codes are the possible class and codes that may be used in a reject PDU that would go into the **reject_resp_info** structure. See **Volume 1— Module 3** for more information on the use of this structure. These codes are defined by the MMS standard, taken from the MMS Protocol Specification ISO 9506.

CODE	MEANING	
REJECT CLASS = 1 CONFIRMED REQUEST PDU PROBLEM		
0	OTHER	Used for errors other than those identified for this Reject PDU type.
1	UNRECOGNIZED-SERVICE	Used when service requested is not supported or recognized.
2	UNRECOGNIZED-MODIFIER	Used when modifier requested is not supported or recognized.
3	INVALID-INVOKE-ID	Used when an Invoke ID does not meet ISO 9506 requirements.
4	INVALID-ARGUMENT	Used when services argument does not meet ISO 9506 requirements.
5	INVALID-MODIFIER	Used when service modifier does not meet ISO 9506 requirements.
6	MAXIMUM-SERVICES-OUTSTANDING -EXCEEDED	Used when negotiated maximum number of outstanding confirmed services is exceeded by a confirmed service request.
7	MAXIMUM-SEGMENT-LENGTH-EXCEEDED (DIS only)	Reserved for further definition per IS specification.
8	MAXIMUM-RECURSION-EXCEEDED	Used when received PDU exceeds maximum data structure nesting level
9	VALUE-OUT-OF-RANGE	Used when received PDU contains one or more parameters whose values exceed the range allowed by ISO 9506.
REJECT CLASS = 2 CONFIRMED RESPONSE PDU PROBLEM		
0	OTHER	Used for errors other than those identified for this Reject PDU type.
1	UNRECOGNIZED-SERVICE	Used when specified service is not supported, or it is not the same service that was requested in the PDU with the specified Invoke ID.
2	INVALID-INVOKE-ID	Used when Invoke ID does not meet ISO 9506 requirements.
3	INVALID-RESULT	Used when service result does not meet ISO 9506 requirements.
4	MAXIMUM-SEGMENT-LENGTH-EXCEEDED (DIS only)	Reserved for further definition per IS specification.
5	MAXIMUM-RECURSION-EXCEEDED	Used when received PDU exceeds negotiated maximum data structure nesting level.
6	VALUE-OUT-OF-RANGE	Used when received PDU contains one or more parameters whose values exceed the range allowed by ISO 9506.

CODE	MEANING	
CLASS = 3 CONFIRMED ERROR PDU PROBLEM		
0	OTHER	Used for errors other than those identified for this Reject PDU type.
1	UNRECOGNIZED-SERVICE	Used when specific service is not supported, not recognized, or not the same service requested in the PDU with the specified Invoke ID.
2	INVALID-INVOKE-ID	Used when Invoke ID does not meet ISO 9506 requirements.
3	INVALID-SERVICE-ERROR	Used when service error does not meet ISO 9506 requirements.
4	VALUE-OUT-OF-RANGE	Used when received PDU contains one or more parameters whose values exceed the range allowed by ISO 9506.
REJECT CLASS = 4 UNCONFIRMED PDU PROBLEM		
0	OTHER	Used for errors other than those identified for this Reject PDU type.
1	UNRECOGNIZED-SERVICE	Used when specified service is not supported, or not recognized.
4	INVALID-ARGUMENT	Used when service argument does not meet ISO 9506 requirements.
8	MAXIMUM-RECURSION-EXCEEDED	Used when received PDU exceeds negotiated maximum data structure nesting level.
9	VALUE-OUT-OF-RANGE	Used when received PDU contains one or more parameters whose values exceed ranged specified by ISO 9506.
REJECT CLASS = 5 PDU ERROR PROBLEM		
0	UNKNOWN-PDU-TYPE	Used when received PDU is not recognized, or not supported.
1	INVALID-PDU	Used when received PDU is syntactically incorrect. Further diagnostics cannot be provided due to severity of error.
2	ILLEGAL-ACSE-MAPPING	Used when received PDU type is not properly mapped to ACSE service primitive. (IS spec only)
REJECT CLASS = 6 CANCEL REQUEST PDU PROBLEM		
0	OTHER	Used for errors other than those identified for this Reject PDU type.
1	INVALID-INVOKE-ID	Used when Invoke ID does not meet ISO 9506 requirements.

CODE	MEANING	
REJECT CLASS = 7 CANCEL RESPONSE PDU PROBLEM		
0	OTHER	Used for errors other than those identified for this Reject PDU type.
1	INVALID-INVOKE ID	Used when Invoke ID does not meet ISO 9506 requirements.
REJECT CLASS = 8 CANCEL ERROR PDU PROBLEM		
0	OTHER	Used for errors other than those identified for this Reject PDU type.
1	INVALID-INVOKE-ID	Used when Invoke ID does not meet ISO 9506 requirements.
2	INVALID-SERVICE-ERROR	Used when service error does not meet ISO 9506 requirements.
3	VALUE-OUT-OF-RANGE	Used when received PDU contains one or more parameters whose values exceed the ranged allowed by ISO 9506.
REJECT CLASS = 9 CONCLUDE REQUEST PDU PROBLEM		
0	OTHER	Used for errors other than those identified for this Reject PDU type.
1	INVALID-ARGUMENT	Used when argument for request does not meet ISO 9506 requirements.
REJECT CLASS=10 CONCLUDE RESPONSE PDU PROBLEM		
0	OTHER	Used for errors other than those identified for this Reject PDU type.
1	INVALID-RESULT	Used when service result does not meet ISO 9506 requirements.
REJECT CLASS=11 CONCLUDE ERROR PDU PROBLEM		
0	OTHER	Used for errors other than those identified for this Reject PDU type.
1	INVALID-SERVICE-ERROR	Used when service error does not meet ISO 9506 requirements.
2	VALUE-OUT-OF-RANGE	Used when received PDU contains one or more parameters whose values exceed the range allowed by ISO 9506.

Appendix B: Opcodes For User Defined Functions

The following tables define the opcode constants used to access the user indication and user confirmation function pointer arrays (**mms_ind_serve_fun** and **mms_conf_serve_fun** respectively). These constants are used as the index into the function pointer arrays. This occurs when you change the functions that get called when an indication or a confirmation is received from the default values set by the library. These are **u_xxxx_ind** and **u_mp_xxxx_conf** or **u_mv_xxxx_conf**.

The following constants are used for the **u_xxxx_ind** and **u_mp_xxxx_conf** functions:

DEFINED CONSTANT	NUMERIC VALUE	SERVICE
MMSOP_STATUS	0	Status
MMSOP_GET_NAMLIST	1	GetNamelist
MMSOP_IDENTIFY	2	Identify
MMSOP_RENAME	3	Rename
MMSOP_READ	4	Read
MMSOP_WRITE	5	Write
MMSOP_GET_VAR	6	GetVariableAccessAttributes
MMSOP_DEF_VAR	7	DefineNamedVariable
MMSOP_DEF_SCAT	8	DefineScatteredAccess
MMSOP_GET_SCAT	9	GetScatteredAccessAttributes
MMSOP_DEL_VAR	10	DeleteVariableAccess
MMSOP_DEF_VLIST	11	DefineNamedVariableList
MMSOP_GET_VLIST	12	GetNamedVariableListAttributes
MMSOP_DEL_VLIST	13	DeleteNamedVariableList
MMSOP_DEF_TYPE	14	DefineNamedType
MMSOP_GET_TYPE	15	GetNamedTypeAttributes
MMSOP_DEL_TYPE	16	DeleteNamedType
MMSOP_INPUT	17	Input
MMSOP_OUTPUT	18	Output
MMSOP_TAKE_CONTROL	19	TakeControl
MMSOP_REL_CONTROL	20	RelinquishControl
MMSOP_DEFINE_SEM	21	DefineSemaphore
MMSOP_DELETE_SEM	22	DeleteSemaphore
MMSOP_REP_SEMSTAT	23	ReportSemaphoreStatus
MMSOP_REP_SEMPOOL	24	ReportSemaphorePoolStatus
MMSOP_REP_SEMENTRY	25	ReportSemaphoreEntryStatus
MMSOP_INIT_DOWNLOAD	26	InitiateDownloadSequence
MMSOP_DOWN_LOAD	27	DownloadSegment
MMSOP_TERM_DOWNLOAD	28	TerminateDownloadSequence
MMSOP_INIT_UPLOAD	29	InitiateUploadSequence
MMSOP_UP_LOAD	30	UploadSegment
MMSOP_TERM_UPLOAD	31	TerminateUploadSequence

DEFINED CONSTANT	NUMERIC VALUE	SERVICE
MMSOP_REQ_DOM_DOWN	32	RequestDomainDownload
MMSOP_REQ_DOM_UPL	33	RequestDomainUpload
MMSOP_LOAD_DOMAIN	34	LoadDomainContent
MMSOP_STORE_DOMAIN	35	StoreDomainContent
MMSOP_DELETE_DOMAIN	36	DeleteDomain
MMSOP_GET_DOM_ATTR	37	GetDomainAttribute
MMSOP_CREATE_PI	38	CreateProgramInvocation
MMSOP_DELETE_PI	39	DeleteProgramInvocation
MMSOP_START	40	Start
MMSOP_STOP	41	Stop
MMSOP_RESUME	42	Resume
MMSOP_RESET	43	Reset
MMSOP_KILL	44	Kill
MMSOP_GET_PI_ATTR	45	GetProgramInvocationAttribute
MMSOP_OBTAIN_FILE	46	ObtainFile
MMSOP_DEF_EC	47	DefineEventCondition
MMSOP_DEL_EC	48	DeleteEventCondition
MMSOP_GET_EC_ATTR	49	GetEventConditionAttributes
MMSOP_REP_EC_STAT	50	ReportEventConditionStatus
MMSOP_ALT_EC_MON	51	AlterEventConditionMonitoring
MMSOP_TRIGGER_EV	52	TriggerEvent
MMSOP_DEF_EA	53	DefineEventActionStatus
MMSOP_DEL_EA	54	DeleteEventAction
MMSOP_GET_EA_ATTR	55	GetEventActionAttributes
MMSOP_REP_EA_STAT	56	ReportEventActionStatus
MMSOP_DEF_EE	57	DefineEventEnrollment
MMSOP_DEL_EE	58	DeleteEventEnrollment
MMSOP_ALT_EE	59	AlterEventEnrollment
MMSOP_REP_EE_STAT	60	ReportEventEnrollmentStatus
MMSOP_GET_EE_ATTR	61	GetEventEnrollmentAttributes
MMSOP_ACK_EVENT_NOT	62	AcknowledgeEventNotification
MMSOP_GET_ALARM_SUM	63	GetAlarmSummary
MMSOP_GET_ALARM_ESUM	64	GetAlarmEnrollmentSummary
MMSOP_READ_JOURNAL	65	ReadJournal
MMSOP_WRITE_JOURNAL	66	WriteJournal
MMSOP_INIT_JOURNAL	67	InitializeJournal
MMSOP_STAT_JOURNAL	68	ReportJournalStatus
MMSOP_CREATE_JOURNAL	69	CreateJournal
MMSOP_DELETE_JOURNAL	70	DeleteJournal
MMSOP_GET_CAP_LIST	71	GetCapabilityList
MMSOP_FILE_OPEN	72	FileOpen

DEFINED CONSTANT	NUMERIC VALUE	SERVICE
MMSOP_FILE_READ	73	FileRead
MMSOP_FILE_CLOSE	74	FileClose
MMSOP_FILE_RENAME	75	FileRename
MMSOP_FILE_DELETE	76	FileDelete
MMSOP_FILE_DIR	77	FileDirectory
MMSOP_ADDL_SERVICE	78	Additional Service
MMSOP_INFO_RPT	79	InformationReport, ucs 0 (**u_info_ind** only)
MMSOP_USTATUS	80	UnsolicitedStatus, ucs 1 (**u_ustatus_ind** only)
MMSOP_EVENT_NOT	81	EventNotification, ucs 2 (**u_evnot_ind** only)
MMSOP_INITIATE	83	Initiate (**u_init_ind** only). See below for **u_mv_init_conf**).
MMSOP_CONCLUDE	84	Conclude
MMSOP_CANCEL	85	Cancel

The following constants are used only for the **u_mv_xxxx_conf** functions in the **mms_conf_serve_fun** function pointer array.

DEFINED CONSTANT	NUMERIC VALUE	FUNCTION
MMSOP_NAMED_READ	90	**u_mv_read_conf** (Named Variable Read)
MMSOP_NAMED_WRITE	91	**u_mv_write_conf** (Named Variable Write)
MMSOP_NAMED_INFO	92	**u_mv_info_conf** (Named InformationReport)
MMSOP_REM_FILE_OPEN	93	**u_mv_fopen_conf** (Remote FileOpen)
MMSOP_REM_FILE_READ	94	**u_mv_fread_conf** (Remote FileRead)
MMSOP_REM_FILE_CLOSE	95	**u_mv_fclose_conf** (Remote FileClose)
MMSOP_INIT_CONN	96	**u_mv_init_conf** (Initiate)
MMSOP_MV_DEF_TYPE	97	**u_mv_deftype_conf** (DefineNamedType)
MMSOP_REM_FILE_COPY	98	**u_mv_fcopy_conf** (Remote FileCopy)
MMSOP_MV_DOWNLOAD	99	**u_mv_termdown_conf** (VM Domain Download)
MMSOP_MV_UPLOAD	100	**u_mv_termupl_conf** (VM Domain Upload)
MMSOP_VM_VAR_READ	101	**u_mv_read_conf** or **u_mv_readvars_conf** (VM Read)
MMSOP_VM_VAR_WRITE	102	**u_mv_write_conf** or **u_mv_writevars_conf** (VM Write)

Appendix C: File Descriptions

After installing MMS-EASE using the separate installation guide supplied with your MMS-EASE product, the files described below will exist on your computer.

Library Files

The following library files must be linked with your application programs to create an executable file using MMS-EASE. The files for Windows systems use a ".lib" extension as shown below. For UNIX systems, the library files have a ".a" extension. For VAX systems, library files have a ".olb" extension.

LIBRARY NAME	DESCRIPTION
mmsease.lib	This is the core MMS-EASE library.
mmsacse.lib	This is the MMS-EASE LLP specific library for environments using ACSE services. This library contains all the ACSE LLP specific functions and data structures needed by MMS-EASE and used within your application program. Only link with this library if you are using MMS-EASE over ACSE services.
mmsepa.lib	This is the EPA specific library. This library contains all the EPA specific functions and data structures needed by MMS-EASE and used within your application program. Only link with this library if you are using MMS-EASE with EPA.
mmsllc.lib	This is the LLC specific library. This library contains all the LLC specific functions and data structures needed by MMS-EASE and used within your application program. Only link with this library if you are using MMS-EASE over LLC.
osiul.lib	This is the OSI Upper Layers Interface library.

Include Files

The following is a list of the include files that may be linked with your files at time of compilation. These files contain the function and structure definitions that the compiler will need when compiling a program that accesses MMS-EASE.

FILE NAME	DESCRIPTION
asn1defs.h	This include file contains the definitions of the ASN1DE functions and data structures. This file is necessary if using the ASN1DE library directly from your application or if logging ASN1 encode or decode.
cfg_util.h	This include file is used with the demonstration program. It provides the necessary defines and function prototypes used to parse the **mmsvar.cfg** and **mms_log.cfg** files supplied with the demo.
fkeydefs.h	These include files are needed by the MMS-EASE demonstration program.

INCLUDE FILES (cont'd)

FILE NAME	DESCRIPTION
gen_list.h	This include file contains definitions required for manipulation of doubly linked circular lists used in various data structures.
glberror.h	This include file contains definitions for all entities that may generate an error code. This file contains the entity code portion of the error code. Error codes are documented in **Volume I — Appendix A**.
glbsem.h	This include file contains definitions and support for multi-threading.
glbtypes.h	This include file contains the global data type definitions. It also contains defines that configure MMS-EASE for the particular type of system on which you are running.
mds.h	This include file contains definitions and declaration for MMS Directory Services (MDS) API.
mem_chk.h	This include file contains declarations for dynamic memory handling functions. It also include bit assignments for memory logging.
mlog_en.h	This include file contains defines to conditionally compile only the necessary SISCO logging functions to optimize size of executable.
mloguser.h	This include file contains the prototypes for Logging Functions for MMS-EASE user code.
mms_defs.h	This include file contains miscellaneous constant and variable definitions, channel states, and MMS Outstanding Request and Indication control structures.
mms_def2.h	This include file contains additional miscellaneous definitions and tables such as MMS Operation Opcodes, parameter limitations, domain state attributes, and miscellaneous data structures.
mms_dfun.h	This include file contains the function declarations necessary to interface with the primitive level decode routines.
mms_err.h	This include file contains the structure and function definitions used to encode and decode MMS-EASE error responses.
mms_llp.h	This include file contains the specific LLP (Lower Layer Provider) definitions for MMS-EASE LLP specific functions and data structures.
mmslog.h	This include file contains MMS-EASE logging macros used internally by MMS-EASE for SISCO Logging (**S_LOG**). See **Volume 1 — Module 2 — Debug Facilities** for more information on **S_LOG**.
mms_log.h	This include file contains MMS-EASE logging definitions in bit assignments. Bit assignments are also provided in **asn1defs.h**, **mem_chk.h**, and **suicacse.h**.
mms_mp.h	This include file contains the common data structures and general function declarations required to interface with MMS-EASE at the primitive level.
mms_ms.h	This include file contains the definitions for the MMS-EASE support and interface functions.
mms_mv.h	This include file contains data structures and functions associated with MMS-EASE virtual machine operations.

INCLUDE FILES (cont'd)

FILE NAME	DESCRIPTION
mms_pxxx.h	These include files contain definitions for the MMS-EASE primitive level functions and data structures for a set of designated services. The xxx indicates one of the following sets of services: mms_pcon.h — Connection Management mms_pdom.h — Domain Management mms_pevn.h — Event Management mms_pfil.h — File Access and Management mms_pjou.h — Journal Management mms_pocs.h — Operation Communication Services mms_pprg.h — Program Invocation Management mms_psem.h — Semaphore Management mms_pvar.h — Variable Access and Management mms_pvmd.h — VMD Management
mms_perr.h	This include file contains data structures and function definitions used for encoding and decoding an error response.
mms_vxxx.h	These include files contain definitions for the MMS-EASE virtual machine functions and data structures for a set of designated services. The xxx indicates one of the following sets of services: mms_vcon.h — Connection Management mms_vdom.h — Domain Management mms_vfil.h — File Access and Management mms_vprg.h — Program Invocation Management mms_vvar.h — Variable Access and Management mms_vvmd.h — VMD Management
mms_ufun.h	This include file contains the MMS-EASE user-defined function declarations. All functions described here must exist in the user application and are invoked as specified events occur. These functions must be written to accomplish the application program objective. It also contains declarations for the operation enable table, opcode print string table, indication service table, and confirmation service table.
mms_user.h	This include file contains all the #include statements needed by your application programs. Use this file to include the individual files because the order of the #include statements is important for proper compilation.
mms_usr.h	This include file is for application programmer use, and calls out all the user accessible include files such as **mms_vvar.h** and **mms_ufun.h**.
scrndefs.h	This include file contains screen control commands needed by the debug versions of the MMS-EASE library and the MMS-EASE demonstration program.
slog.h	This include file contains general declarations and defines for the **S_LOG** (SISCO Logging) system.
stime.h	This include file contains functions for handling time in the **S_LOG** system.
suicacse.h	This include file contains all the function and data structure definitions needed for ACSE versions of SUIC. This file will only need to be included if using ACSE level SUIC functions from within your program or if logging SUIC level operations.

INCLUDE FILES (cont'd)

FILE NAME	DESCRIPTION
sysincs.h	This file contains definitions and include statements for selecting which system-specific include files to select for a given operating environment.
xxxx_a.h	These are include files that contain OSI Stack-specific definitions. As of this writing, the following are valid include files of this type:

FILE NAME	DESCRIPTION
decnet_a.h	For DECnet
marben_a.h	For SISCO OSI Stack

disptmpl.h lber.h ldap.h msdos.h proto-ld.h proto-lb.h srchpref.h	These include files pertain to support for LDAP.

MMS-EASE Demonstration Program Files

Included with MMS-EASE product are the source files used to create two MMS-EASE demonstration programs. These files can be examined to see how an actual application can be put together using MMS-EASE. Some files may be of importance since they provide a means for creating subsets of MMS-EASE. In addition, some files contain general purpose input routines suitable for use in your application programs. See **Appendix F — MMS-EASE Demonstration Programs** for more information.

Standard MMS-EASE Demo Files

FILE NAME	DESCRIPTION
datademo.c	This file documents how variable access is used for client and server including alternate access usage and low level data conversion.
dataval.c	This file contains only a stub function which may be modified or replaced with additional 'test' code for data validation.
getval5.c fkey.c	These source files contain keyboard and display control routines used by the debug versions of the MMS-EASE libraries and the demonstration program. You will need to compile and link these to use the debug libraries and to create a demonstration program. The **fkey.c** file may have a slightly different name, dependent on the operating system being used. These are as follows:

NAME	DESCRIPTION
fkey.c	For MS-DOS on IBM-PC/XT/AT compatibles
mvkey.c	For VMS on DEC Alpha computers
ntkey.c	For WindowsNT and Windows95
ukey.c	For UNIX Systems such as IS UNIX, Solaris, AIX
wkey.c	For Windows 3.x

DEMO FILES (cont'd)

FILE NAME	DESCRIPTION
mmsaxxx.c	Files of this form are source code files for the demonstration program included with your MMS-EASE. xxx indicates the type of services that are accessed within the module. These source files can be modified, compiled, and linked with the MMS-EASE libraries to create your own version of SISCO's demonstration program. The **xxx** indicates the type of services that are accessed within the module:
	mmsapp.c — main program loop of the demo including examples for logging, and event handling
	mmsaacse.c — deals with MMS SUIC services
	mmsacon.c — deals with Connection Management Services
	mmsadom.c — deals with Domain Management Services
	mmsaevn.c — deals with Event Management Services
	mmsafil.c — deals with File Access and Management Services
	mmsajou.c — deals with Journal Management Services
	mmsallc.c — deals with LLC services
	mmsallp.c — deals with LLP services
	mmsamisc.c — deals with miscellaneous MMS services such as how to read in the **mms_log.cfg** file
	mmsaocs.c — deals with Operator Communication
	mmsaprg.c — deals with Program Invocation Management
	mmsasem.c — deals with Semaphore Management
	mmsavar.c — deals with Variable Access and Management
	mmsavmd.c — deals with VMD Management
mmsatest.c	This file contains only a stub function which may be modified or replaced with additional 'test' code.
mmsdemo.cfg	This is the general configuration file for the MMS-EASE demonstration program.
mmsdemo.txt	This is an ASCII file containing documentation on how to use the MMS-EASE demonstration software.
mms_log.cfg	This is the Logging Configuration File. The purpose of this file is to enable various MMS-EASE log levels for use with the MMS-EASE demo.
mmsop_en.c	This file needs to be compiled and linked with your application programs regardless of whether you are creating a subset of MMS-EASE or not. See **Volume 1 — Module 2** for more information on subset creation.
mmsop_en.h	This include file contains the definitions needed to enable specific MMS services and contains the opcode definitions for the user definable function pointer arrays. This file should be modified so that it only contains those services that are needed by your application to prevent linking services that are not needed. See **Volume 1 — Module 2** for more information on subset creation.
mmsvar.cfg	This configuration file allows MMS objects to be added to the MMS-EASE Demo such as non-standard types, named variables, program invocations, and others.

DEMO FILES (cont'd)

FILE NAME	DESCRIPTION
u_acse.c	This file contains the application functions called by SUIC when communication events occur for ACSE. These functions are mostly for user information and not typically required. In some applications, it provides a mechanism to print ACSE transfer information, if enabled.
u_confrm.c	This file contains user confirm functions invoked from **`mms_req_serve`**. In addition, there is a set of files with the form, **u_cxxx.c**. These contain the information on confirm functions and data structures for each individual service. The **xxx** represents one of the following services: u_ccon.c — deals with Connection Management u_cdom.c — deals with Domain Management u_cevn.c — deals with Event Management u_cfil.c — deals with File Access and Management u_cjou.c — deals with Journal Management u_cocs.c — deals with Operation Communications u_cprg.c — deals with Program Invocation Management u_csem.c — deals with Semaphore Management u_cvar.c — deals with Variable Access and Management u_cvmd.c — deals with VMD Support
u_data.c	This file contains the user defined data handling functions such as **`u_get_named_addr`** and **`ms_add_named_var`**.
u_indic.c	This file contains user defined code to service indications invoked from mms_ind_serve. In addition, there is a set of files with the form, **u_ixxx.c**. These contain the information on indication functions and data structures for each individual service. The **xxx** represents one of the following services: u_icon.c — deals with Connection Management u_idom.c — deals with Domain Management u_ievn.c — deals with Event Management u_ifil.c — deals with File Access and Management u_ijou.c — deals with Journal Management u_iocs.c — deals with Operation Communications u_iprg.c — deals with Program Invocation Management u_isem.c — deals with Semaphore Management u_ivar.c — deals with Variable Access and Management u_ivmd.c — deals with VMD Support
u_llc.c	This file contains the user defined LLC indication functions.
userdefs.h	This file contains user application definitions only used for the demo program such as how to turn off logging and use the binary search option.
uservar.c	This file contains miscellaneous user variable declaration and type definition initialization information.

In addition to the source files, each product supplies a make or project file used to build the MMS-EASE demo program. Please consult the product's installation and configuration guide for more information.

MMSEASY Demo Files

mmseasy.c	This is the only source code file used for MMSEASY, a simple MMS-EASE application, all in one source module. It contains code for both a client and a server.
mmsop_en.c	This file needs to be compiled and linked with your application programs
mmsop_en.h	This include file contains the definitions needed to enable specific MMS services and contains the opcode definitions for the user definable function pointer arrays.

In addition to the source files, each product supplies a make or project file used to build the MMSEASY demo program. Please consult the product's installation and configuration guide for more information.

Appendix D. Service vs. Function Naming Conventions

MMS-EASE uses abbreviations that are designated "**xxxx**" for the various functions and data structures related to a given MMS service. These abbreviations are shown below along with the corresponding MMS service.

SERVICE NAME	CORRESPONDING "xxxx" DEFINITION
AcknowledgeEventNotification	ackevnot
AlterEventConditionMonitoring	altecm
Cancel	cancel
Conclude	conclude
CreateProgramInvocation	crepi
DefineEventAction	defea
DefineEventCondition	defec
DefineEventEnrollment	defee
DefineNamedType	deftype
DefineNamedVariable	defvar
DefineNamedVariableList	defvlist
DefineSemaphore	defsem
DeleteDomain	deldom
DeleteEventAction	delea
DeleteEventCondition	delec
DeleteEventEnrollment	delee
DeleteNamedType	deltype
DeleteNamedVariableList	delvlist
DeleteProgramInvocation	delpi
DeleteSemaphore	delsem
DeleteVariableAccess	delvar
DownloadSegment	download
EventNotification	evnot
FileClose	fclose
FileDelete	fdelete
FileDirectory	fdir
FileOpen	fopen
FileRead	fread
FileRename	frename
GetAlarmEnrollmentSummary	getaes
GetAlarmSummary	getas
GetCapabilityList	getcl
GetDomainAttributes	getdom
GetEventActionAttributes	geteaa

SERVICE NAME	CORRESPONDING "xxxx" DEFINITION
GetEventConditionAttributes	geteca
GetEventEnrollmentAttributes	geteea
GetNamedTypeAttributes	gettype
GetNameList	namelist
GetNamedVariableList	getvlist
GetProgramInvocationAttributes	getpi
GetVariableAccessAttributes	getvar
Identify	ident
InformationReport	info
InitializeJournal	jinit
Initiate	init
InitiateDownloadSequence	initdown
InitiateUploadSequence	initupl
Input	input
Kill	kill
LoadDomainContent	loaddom
ObtainFile	obtfile
Output	output
Read	read
ReadJournal	jread
Reject	reject
Rename	rename
RelinquishControl	relctrl
ReportEventConditionStatus	repecs
ReportEventEnrollmentStatus	repees
ReportJournalStatus	jstat
ReportPoolSemaphoreStatus	rspool
ReportSemaphoreEntryStatus	rsentry
ReportSemaphoreStatus	rsstat
RequestDomainDownload	rddwn
RequestDomainUpload	rdupl
Reset	reset
Resume	resume
Start	start
Status	status
Stop	stop
StoreDomainContent	storedom
TakeControl	takectrl
TerminateDownloadSequence	termdown
TerminateUploadSequence	termupl
TriggerEvent	trige

SERVICE NAME	CORRESPONDING "xxxx" DEFINITION
UnsolicitedStatus	ustatus
UploadSegment	upload
Write	write
WriteJournal	jwrite

Appendix E. Global Data Type Definitions

The MMS-EASE data structures use variable type definitions that are defined in the include file **glbtypes.h**. This provides for a portable implementation by hiding the mapping of a specific variable type on a given machine from the programs themselves.

#define ST_CHAR char

This type defines storage for an 8-bit entity. Usually, this type of element is used to store character strings and octets.

#define ST_INT signed int

This type defines storage for a signed integer. The precision of this type of element will vary according to the compiler.

#define ST_LONG signed long

This type, defines storage for a signed long integer. The precision of this type of element will vary according to the compiler.

#define ST_UINT unsigned int

This type defines storage for an unsigned integer. The precision of this type of element will vary according to the compiler.

#define ST_ULONG unsigned long

This type, defines storage for an unsigned long integer. The precision of this type of element will vary according to the compiler.

#define ST_UCHAR unsigned char

This type, ST_UCHAR, defines storage for an unsigned 8-bit entity. It is used to store an unsigned numeric value between 0 and 255 or a BOOLEAN variable.

#define ST_BOOLEAN unsigned char

This type defines storage for an unsigned 8-bit entity, in which 0 = SD_FALSE. All other values are considered to be SD_TRUE.

#define ST_INT8 signed char

This type, defines storage for a 8-bit entity. It is used to store a signed numeric value between -128 and 127.

#define ST_UINT8 unsigned char

This type defines storage for a 8-bit entity. It is used to store an unsigned numeric value between 0 and 255.

#define ST_INT16 signed short

This type defines storage for a 16-bit entity. It is used to store a signed numeric value between -32,768 and 32,767.

#define ST_UINT16 unsigned short

This type defines storage for an unsigned 16-bit entity. It is used to store a numeric value between 0 and 65,535.

```
#define ST_INT32 signed long
```

This type defines storage for a 32-bit entity. It is used to store a signed numeric value between -2^{31} and $+2^{31}-1$.

```
#define ST_UINT32 unsigned long
```

This type defines storage for an unsigned 32-bit entity. It is used to store an unsigned numeric value between 0 and 2^{32}.

```
#define ST_FLOAT float
```

This type defines storage for a four byte single precision floating point number.

```
#define ST_DOUBLE double
```

This type defines storage for an eight byte double precision floating point number.

```
#define ST_VOID void
```

This defines a function return value for functions that do not return any values.

```
#define SD_FALSE    0
```

This defines a value of FALSE for boolean variables.

```
#define SD_TRUE     1
```

This defines a value of TRUE for boolean variables.

```
#define SD_SUCCESS  0
```

This defines a return value of success for functions that return values of type ST_RET.

```
#define SD_FAILURE  1
```

This defines a return value of failure for functions that return values of type ST_RET.

Appendix F. MMS-EASE Demonstration Programs

The MMS-EASE product provides source code for two demonstration programs that can be compiled and linked with the supplied MMS-EASE libraries to create ready-to-run applications.

1. Standard MMS-EASE Demonstration Program — This program consists of many source modules. Its menu driven interface can be used to illustrate all the MMS services supported by MMS-EASE. It also contains sample code for using events on an operating system basis.
2. MMSEASY — This simple MMS-EASE application is contained all in one source module. There is code for both a client and a server. This code is intended only as a simple demonstration of what can be done easily, and does not make use of system specific event notification facilities provided by MMS-EASE.

Standard MMS-EASE Demonstration Program

The MMS-EASE demo code provides examples of how MMS services and functionality can be implemented. After MMS-EASE has been properly installed using the supplied installation guide, the demo can then be created. Usually a sample makefile or batch/command file is supplied with your product. Making the demo is a useful test to ensure that your environment, compiler and linker are properly configured to run MMS-EASE. Please check the System Requirements section of your installation guide if you get any error messages during compiling and linking to make sure that you have the proper software components. Please note the following about the demo program:

1. You MUST use the **mmsop_en.h** and **mmsop_en.c** files supplied by the current product. The include file should be modified to correctly match the services to be included in the demo. For some systems with memory limitations, it may not be possible to enable all the services. The demo program uses the constants defined in **mmsop_en.h** to control the conditional compilation of the source so that references to unused functions are not made.
2. In the makefile for the MMS-EASE demo, there are symbols used to control the compilation of the demonstration program. You can change the way the conditional compilation of the demo program works by using these compile switches.

   `DEBUG_SISCO` If this symbol is defined, the demo program will use the **`x_chk`** versions of the SISCO memory management API.

   `MAP30_ACSE` You must define this symbol in order to use the ACSE LLP.
3. The documentation on the setup and operation of the demonstration program can be found on your product diskette or tape or if not, by contacting SISCO Technical Support. This ASCII file is called **mmsdemo.txt**.
4. In addition to the **mmsop_en.h** file, there are several configuration files used to run the demo. These are described in detail in the sections to follow.

The General Demo Configuration File (mmsdemo.cfg)

This file contains information required to properly configure the demo after powering up. The default file is named **mmsdemo.cfg** and resides in the same directory as the demo source files. It automatically performs the process of activating, registering, and posting listens to specific channels for the demo program. In addition, it can be used to set defaults for initiate parameters such as the maximum message size and total number of channels. This is an ASCII file that can be modified using any text editor and contains the following information:

#1 CHANNEL CONFIGURATION

TOTAL_CHAN This indicates the total number of channels to be supported for this application.

The rest of these parameters are optional.

MAX_MSG_SIZE This indicates the maximum PDU message size.

MAX_NUM_VARS This specifies the maximum number of named variables to be defined for the demo.

MAX_NUM_TYPES This specifies the maximum number of named types to be defined for the demo.

MAX_NUM_DOMS This specifies the maximum number of domains to be defined for the demo.

MAX_NUM_PIS This specifies the maximum number of program invocations to be defined for the demo.

#2 NAMES TO BE ACTIVATED

This is a list of all the local AR Names that need to be activated for the demo.

#3 NAMES TO BE REGISTERED ON CHANNELS

This assigns a previously activated AR Name (listed in Section #2) to be registered on a channel. Registration is sequential on channel number beginning with channel 0.

#4 CHANNELS TO BE LISTENING (CALLED)

This indicates the channels that will be used for incoming connections.

In the sample file shown below, a blank line or any line beginning with a pound sign (**#**) denotes the beginning of an entry. All data following the **#** sign on that line is ignored. This can be used to enter comments. It is possible to save on keystrokes in this file by configuring a range of channels for Sections #3 and #4. The syntax for a channel range is [{low_channel}..{high_channel}]. An example using channel ranges is shown below:

```
#1 CHANNEL CONFIGURATION
TOTAL_CHAN=64
MAX_MSG_SIZE=65535
MAX_NUM_VARS=10000
MAX_NUM_TYPES=10000
MAX_NUM_DOMS=10000
MAX_NUM_PIS=10000
#2 NAMES TO BE ACTIVATED
Local1
#3 NAMES TO BE REGISTERED ON CHANNELS
Local1[0-63]
#4 CHANNELS TO BE LISTENING (CALLED)
[32-63]
#5 END OF FILE
```

The MMS Object Configuration File (mmsvar.cfg)

An additional configuration file, used with the demo program, allows adding MMS objects to the MMS-EASE object database. This file is also read in upon power up. The default file is named **mmsvar.cfg** and resides in the same directory as the demo source files. It is an ASCII file that can be modified using almost any text editor and contains the following information:

1. DOMAIN SECTION

Each NamedDomain is formed from two keywords: 1) **`DomainName =`** and 2) **`Capability =`**. Spaces, Tabs, and the LineJoin character '|' may be used to improve readability. A brief description of each keyword is listed below:

`DomainName =` This is the name of a MMS Domain visible on the network as a NamedDomain object. All entries found in this section are entered into the default VMD of the MMS-EASE Demo.

`Capability =` This is a one word capability string added to the list of capabilities for the Domain being configured. A capability list is optional for a Domain. There may also be more than one capability associated with a domain. The list of capabilities for a domain is terminated by the beginning of the next DomainName or by the end of the Domain Section.

2. PROGRAM INVOCATION SECTION

Each ProgramInvocation is formed from two keywords: 1) **`PiName =`** and 2) **`Domain =`**. Spaces, Tabs, and the LineJoin character '|' may be used to improve readability. A brief description of each keyword is listed below:

`PiName =` This is the name of a MMS ProgramInvocation visible on the network as a ProgramInvocation Object. All entries found in this section are entered into the default VMD of the MMS-EASE demo.

`Domain =` This is the name of a NamedDomain object associated with the Program being configured. A list of Domains is optional for a ProgramInvocation. There may be more than one Domain referenced by a ProgramInvocation. The Domains used in this keyword must already be pre-configured in the Domains Section or be created by the application. The list of domains is terminated by the beginning of the next PiName or by the end of the Program Invocation Section.

3. TYPES SECTION

Each complex MMS Type is formed from two keywords: 1) **`TypeName =`** and 2) **`TypeDef =`**. Spaces, Tabs, and the LineJoin character '|' may be used to improve readability. A brief description of each keyword is listed below:

`TypeName =` This is the name of a MMS Type visible on the network as a NamedType object The name may be up to 32 characters corresponding to the rules of an ISO Identifier. All entries found in this section are entered into the VMD of the MMS-EASE demo.

`TypeDef =` This is a MMS NamedType Definition as described using MMS Type Definition Language (TDL). See **Volume II — Module 5** for more information on TDL.

4. VARIABLES SECTION

Each Named Variable is formed from two keywords: 1) **`VarName =`** or **`VarNameInit =`** and 2) **`TypeName =`**. Spaces, Tabs, and the LineJoin character '|' may be used to improve readability. A brief description of each keyword is listed below:

`VarName =` This is the name of a MMS Variable visible on the network as a NamedVariable object. The name may be up to 32 characters corresponding to the rules of an ISO Identifier. Entries found in this section are added by default to the default VMD of the MMS-EASE demo. Variables may also be added to a Domain that has be created by the Domain section by prefixing the name of the domain to the name of the variable. The period character (.) is required when adding a Domain-specific variable to separate the name of the Domain from the name of the Variable. Data space for the variable is allocated by the demo program and initialized to default values of all zeros (0).

`VarNameInit =` This is the same as the VarName keyword except that the user will be prompted to enter the data for this variable after the variable is added to the database.

`TypeName =` This is the name of a NamedType object associated with the variable being configured. The name that follows this keyword must be defined in the Types Section of this file.

5. NAMED VARIABLE LISTS SECTION

Each NamedVariableList is formed from the following keywords: 1) **`VarList =`** or **`VarNameInit =`**, 2) **`Domain =`**, 3)**`VarName =`**. Spaces, Tabs, and the LineJoin character '|' may be used to improve readability. There may be only one domain entry for the NamedVariableList. This implies that not only is entire list domain-specific, but that all the NamedVariables referenced in the list by **`VarName`** = also must exist in the same domain. A brief description of each keyword is listed below:

`VarList =` This is the name of a list of MMS Variables visible on the network as a NamedVariableList object. The name may be up to 32 characters corresponding to the rules of an ISO Identifier. Entries found in this section are added by default to the default VMD of the MMS-EASE demo. Variables may also be added to a Domain from Domain Section by including the **`Domain`**= keyword in the configuration.

`Domain =` This is the name of a NamedDomain object associated with the NamedVariableList being configured. The name found after this keyword must be created by the application or configured in the Domain section of this file.

`VarName =` This is the name of a NamedVariable object that must exist in the same scope (VMD or Domain) as the NamedVariableList being configured. The name found after this keyword must be created by the application or be configured in the Variables Section of this file.

`VarListEnd` This keyword terminates the configuration for a NamedVariableList and causes the object to be created.

In the file shown below, **%xxxBegin** denotes the beginning of a section. **%xxxEnd** denotes the end of a section. All data following the **#** sign on a line is a comment and can be ignored.

```
#
# ------- SECTION: DomainSection
#

%DomainsBegin
  DomainName = DadsDomain | Capability = Lift | Capability = Haul
                       Capability = Drink | Capability = Eat

  DomainName = ChildsDomain | Capability = Play | Capability = School
                         Capability = Sleep

  DomainName = MomsDomain | Capability = Shop | Capability = Phone
                       Capability = Swim

  DomainName = DomainOfTheUnsigned
  DomainName = DomainNamed1
  DomainName = DomainNamed2
  DomainName = DomainNamed3
  DomainName = DomainNamed4
  DomainName = DomainNamed5
%DomainsEnd

#
# ------- SECTION: ProgramInvocationSection
#

%PisBegin
  PiName = FamilyFunctionPI
  Domain = DadsDomain
  Domain = MomsDomain
  Domain = ChildsDomain

  PiName = EmptyPI

  PiName = GTWY_PI
  Domain = Robot
  Domain = Flex_Gage

  PiName = FivePartPI
  Domain = DomainNamed1
  Domain = DomainNamed2
  Domain = DomainNamed3
  Domain = DomainNamed4
  Domain = DomainNamed5

%PisEnd

#
```

```
# ------- SECTION: TypesSection
#

%TypesBegin

  TypeName = cc_dispatch_typ  | TypeDef = {Ushort, Ubyte}
  TypeName = surge_status_typ | TypeDef = {Ubyte, [7:Bool]}

       # struct mixed_int
       #  {
       #  BOOLEAN  b;
       #  SHORT    s;
       #  LONG     l;
       #  };

  TypeName = mixed_int   | TypeDef = {(b)Bool,(s)Short,(l)Long}

       # struct mixed_int2
       #   {
       #   BOOLEAN  b;
       #   LONG     l;
       #   SHORT       s;
       #   };


  TypeName = mixed_int2    | TypeDef = {(b)Bool,(l)Long,(s)Short}

  TypeName = arr_5_short | TypeDef = [5:Short]
  TypeName = dim2_arr      | TypeDef = [2:[3:Short]]

       # struct oct_str
       #   {
       #   BOOLEAN  b1;
       #   UBYTE ostr1[3]; /* fixed length octet string           */
       #   BOOLEAN  b2;
       #   SHORT ostr2_len;      /* variable length octet string  */
       #   UBYTE ostr2[7];
       #   LONG l;
       #   };

  TypeName = oct_Str       | TypeDef =
{(b1)Bool,(ostr1)[3:Ubyte],(b2)Bool,(ostr2_len)Short,(ostr2)[7:Ubyte],(l)Long}

       # struct bit_str
       #   {
       #   BOOLEAN  b1;
       #   UBYTE bstr1[3]; /* fixed length (24) bit string  */
       #   BOOLEAN  b2;
       #   UBYTE bstr2[6];
       #   LONG l;
       #   };
```

```
  TypeName = bit_str       | TypeDef =
{(b1)Bool,(bstr1)Bstring24,(b2)Bool,(bstr2)Bstring48,(l)Long}

      # struct vis_str
      #   {
      #   BOOLEAN  b1;
      #   UBYTE vstr1[6]; /* Fixed length visible string   */
      #   BOOLEAN  b2;
      #   UBYTE vstr2[11];      /* Fixed length visible string   */
      #   LONG l;
      #   };

  TypeName = vis_str       | TypeDef =
{(b1)Bool,(vstr1)Vstring5,(b2)Bool,(vstr2)Vstring10,(l)Long}

      # struct flt_str
      #   {
      #   BOOLEAN  b1;
      #   FLOAT f;        /* 4 byte float                        */
      #   BOOLEAN  b2;
      #   DOUBLE d;       /* 8 byte float                        */
      #   };

  TypeName = flt_str       | TypeDef = {(b1)Bool,(f)Float,(b2)Bool,(d)Double}

      # struct bcd_str
      #   {
      #   BYTE bcd2;            /* 2 digit BCD                          */
      #   SHORT bcd4;           /* 4 digit BCD                          */
      #   LONG bcd5;            /* 5 digit BCD                          */
      #   LONG bcd6;            /* 6 digit BCD                          */
      #   SHORT bcd3;           /* 3 digit BCD                          */
      #   LONG bcd7;            /* 7 digit BCD                          */
      #   LONG bcd8;            /* 8 digit BCD                          */
      #   BYTE bcd1;            /* 1 digit BCD                          */
      #   };

  TypeName = bcd_str       | TypeDef =
{(bcd2)Byte,(bcd4)Short,(bcd5)Long,(bcd6)Long,(bcd3)Short,(bcd7)Long,(bcd8)Long,(
bcd1)Byte}
```

```
      # struct time_str
      #   {
      #   BOOLEAN  b1;
      #   LONG btime1;          /* Binary Time Of Day - 4 byte   */
      #   BOOLEAN  b2;
      #   LONG btime2;          /* Binary Time Of Day - 6 byte   */
      #   BOOLEAN  b3;
      #   LONG gtime;           /* Generalized Time                    */
      #   };
```

~~TypeName = time_str | TypeDef = {(b1)Bool,(btime1)Long,(b2)Bool,(btime2)Long,(b3)Bool,(gtime)Long}~~

```
        # struct mixed_int
        #   {
        #   BOOLEAN  b;
        #   SHORT     s;
        #   LONG     l;
        #   }[3];

  TypeName = arr_str1    | TypeDef = [3:{(b)Bool,(s)Short,(l)Long}]

        # struct
        #   {
        #   LONG l;
        #   struct
        #     {
        #     UBYTE ub1;
        #     ULONG ul;
        #     } arr_str[10];
        #   UBYTE ub2;
        #   };
  TypeName = etc1_typ      | TypeDef =  { Short, Short, Long, Double, Bool, Bool,
Ushort, Ulong, Vstring16, Vstring80, Ubyte, Ubyte }
  TypeName = etc2_typ      | TypeDef =  { Vstring16, Vstring80, Long, Ulong,
Bool, { Short, Short, Vstring16 }, { Long, Short, Vstring80 } }
  TypeName = etc3_typ      | TypeDef =  { Bool, Short, Long, Bool, Long, Short }
  TypeName = nest4   | TypeDef =  { Ubyte, { Long, { Ubyte, { Long, { Ubyte } } } }
} }
  TypeName = complex1      | TypeDef =
{(l)Long,(arr_str)[10:{(ub1)Ubyte,(ul)Ulong}],(ub2)Ubyte}
  TypeName = VarOstr10     | TypeDef =  OVstring10
  TypeName = VarBstr20     | TypeDef =  BVstring20
  TypeName = Bstr20  | TypeDef =  Bstring20
  TypeName = exstr1  | TypeDef =  {comp1)Short,(comp2)Short,(comp3)Short}
  TypeName = exstr2  | TypeDef =  {(comp4)Long,(comp5)Long,(comp6)Long}
  TypeName = exstr_2       | TypeDef =  {(arr1)[23:Short],(arr2)[41:Short]}
  TypeName = arr2dim       | TypeDef =  [5:[10:Short]]
  TypeName = str1    | TypeDef =
{(l)Long,(str2){(arr_str)[10:{(ub)Ubyte,(ul)Ulong}]}}
  TypeName = str3    | TypeDef =
{(l)Long,(arr_str)[10:{(ub1)Ubyte,(ul)Ulong}],(ub2)Ubyte}
  TypeName = arr     | TypeDef =  [10:Short]
  TypeName = arr_str       | TypeDef =
[5:{(l)Long,(comp1)[5:{(ub1)Ubyte,(ul1)Ulong}],(comp2)[5:{Ubyte,(ul2) Ulong}]}]
  TypeName = arr_str2      | TypeDef =
{(arr_str1)[5:{(arr_str11)[5:{(ub1)Ubyte,(ul1)Ulong,}],(ubl1)Ulong}],
(arr_str2)[5:{(ub2)Ubyte,(ul2)Ulong}]}

%TypesEnd
```

```
#
# ------- SECTION: VarsSection
#

%VarsBegin

  VarName = DomainOfTheUnsigned.UnsignedOctet | TypeName = Unsigned8
  VarName = DomainOfTheUnsigned.UnsignedWord | TypeName = Unsigned16
  VarName = DomainOfTheUnsigned.UnsignedDWord | TypeName = Unsigned32
  VarName = arr_str8             | TypeName = arr_str8
  VarName = a$$dollarsign       | TypeName = Integer16
  VarName = cc_job_Var           | TypeName = cc_dispatch_typ
  VarName = pacman_surge  | TypeName = surge_status_typ
  VarName = mixed_int     | TypeName = mixed_int
  VarName = mixed_int2           | TypeName = mixed_int2
  VarName = arr_5_short   | TypeName = arr_5_short
  VarName = dim2_arr             | TypeName = dim2_arr
  VarName = oct_str       | TypeName = oct_Str
  VarName = bit_str       | TypeName = bit_str
  VarName = vis_str       | TypeName = vis_str
  VarName = flt_str       | TypeName = flt_str
  VarName = time_str      | TypeName = time_str
  VarName = arr_str1      | TypeName = arr_str1
  VarName = complex1             | TypeName = complex1
  VarName = etc1                 | TypeName = etc1_typ
  VarName = etc2                 | TypeName = etc2_typ
  VarName = etc3                 | TypeName = etc3_typ
  VarName = nest4                | TypeName = nest4

  VarName = VarOstr10            | TypeName = VarOstr10
  VarName = VarBstr20            | TypeName = VarBstr20
  VarName = Bstr20        | TypeName = Bstr20
  VarName = exstr1        | TypeName = exstr1
  VarName = exstr2        | TypeName = exstr2
  VarName = arr                  | TypeName = arr
  VarName = exstr_2       | TypeName = exstr_2
  VarName = arr2dim       | TypeName = arr2dim
  VarName = str1                 | TypeName = str1
  VarName = str3                 | TypeName = str3
  VarName = arr_str       | TypeName = arr_str
  VarName = arr_str2             | TypeName = arr_str2

%VarsEnd
```

```
#
# ------- SECTION: VarListSection
#

%VarListsBegin

  VarList = FlexVars | Domain=Flex_Gage | VarName = Z_Dimension | VarName =
Y_Dimension | VarName = X_Dimension | VarListEnd
  VarList = PopularEtcs | VarName = etc1 | VarName = etc2 | VarListEnd
  VarList = RobotVars | Domain=Robot | VarName = Z_Axis | VarName = Y_Axis |
VarName = X_Axis | VarListEnd

  VarList = AlmostEveryone |
          VarName = cc_job_Var  |
          VarName = pacman_surge|
          VarName = mixed_int   |
          VarName = mixed_int2  |
          VarName = arr_5_short |
          VarName = dim2_arr           |
          VarName = oct_str            |
          VarName = bit_str            |
          VarName = vis_str            |
          VarName = flt_str            |
          VarName = time_str    |
          VarName = arr_str1    |
          VarName = complex1           |
          VarName = etc1        |
          VarName = etc2        |
          VarName = etc3        |
          VarName = nest4       | VarListEnd

%VarListsEnd
```

The MMS Logging Configuration File (mms_log.cfg)

An additional configuration file is used to configure MMS-EASE log levels to be used with the demo This file is also read in upon power up. This is done in the MMS-EASE demo source file **mmsamisc.c**. The default log configuartion file is named **mms_log.cfg** and resides in the same directory as the demo source files.

Data values must immediately follow the '=' and be in the required format, or they will be ignored. Keywords are not case sensitive, and can be present in any order. Lines that begin with '#' or do not contain a recognized keyword are ignored. This is an ASCII file that can be modified using any text editor. Below is the default **mms_log.cfg** file supplied with the MMS-EASE demo source code:

```
#########################################################################
#  COMPONENT_NAME: MMS-EASE Logging Configuration File mms_log.cfg
#
#     This file is used to control the amount of logging performed by
#     the MMS-EASE Debug Libraries. Several keywords are possible. All
#     are listed and described below:
#
#     The parameters listed below are valid and are used to control the
#     format of the Log file.
#
#     FileLogEn -Logged messages will go to a file as opposed to stdout.
#
```

```
#      MemLogEn - Some number of log messages will accumulate in memory
#                 before being announced. This is useful for gathering
#                 timing info.
#
#      TimeEnable= This parameter has 3 possible values. Set this
#                 parameter to '0' to disable the time stamp completely,
#         '1' for standard time/date logging, or '2' for elapsed time.
#      HardFlush -   This will cause the file to be closed and reopened
#                 after each message is sent to it. This is for very
#                 untrusting users.
#
#      NoAppend -    Any existing log files will be overwritten.
#
#      NoMsgheader - Suppresses the message identifier.
#
#      NoWipe -      The wipe bar which shows the location where the
#                 file wrapped. Will NOT appear in the file.
#
#      NoWrap -   Will prevent the file from wrapping. Once the
#                logfile has reached its upper boundary size, additional
#                 messages are discarded.
#
#      Setbuf -    Logged messages are always output without being
#                 buffered.
#
#      LogFileSize= When the parameter'FileLogEn'is configured,this value
#                represents the size of the circular log file in bytes.
#                There can be no ',' or radix point in the value.
#      LogFileName= When the parameter 'FileLogEn' is configured, this is
#                 the value of an alternate log file name. Only change
#                 this if you don't want your log file placed in the
#                 default directory.
#
#      LogMemItems=  When the parameter 'MemLogEn' is configured, this is
#                 how many items to buffer in memory before sending them
#                to a file. There can be no ',' or radix point in the
#                value.
#
#      DumpFileName= When the parameter 'MemLogEn' is configured, this is
#                 the value of an alternate log file name. Only change
#                    this if you don't want your log file placed in the
#                    default directory.
#
#      MemAutoDump - When Memory Logging is enabled this is an append and
#                   clear feature.
#
# ------- General Error Logging
#
#      MEM_LOG_ERR -Memory misuse, buffer overwrites, multiple frees,etc.
#      ASN1_LOG_ERR - ASN.1 tools misuse
#      ASN1_LOG_NERR - Recoverable ASN.1 errors ( bad grammar )
#      ACSE_ERR_PRINT - SUIC state machine problems and LLP errors
#      MMS_LOG_ERR - MMS state machine problems
#      MMS_LOG_NERR - Recoverable MMS errors ( bad grammar )
#
# ------- Dynamic Memory Tracking( good for leaks and multiple frees )
#
#      MEM_LOG_CALLOC - FILE and LINE of where buffers are calloced
#      MEM_LOG_MALLOC - FILE and LINE of where buffers are calloced
#      MEM_LOG_REALLOC - FILE and LINE of where buffers are realloced
#      MEM_LOG_FREE - FILE and LINE of where buffers are freed
#
# ------- ASN.1 Parsing
#
#      ASN1_LOG_DEC - Parses class, form, tag, length, and contents
```

```
#      ASN1_LOG_ENC - Shows encoding of a PDU at the ASN.1 level
#
# ------- Application Level Logging
#
#      MMSCONF_PRINT - Prints MMS-Confirmations at application level only
#      MMSIND_PRINT - Prints MMS-Indications at application level only
#
# ------- MMS Transaction Object Handling
#
#      MMS_LOG_DEC - Shows states used in the parse of an MMS PDU
#      MMS_LOG_ENC - Shows encoding of an MMS PDU
#      MMS_LOG_ACSE - Audits MMS events passed up through the ACSE layer
#      MMS_LOG_LLC - Audits MMS events passed up through the LLC layer
#      MMS_LOG_IQUE - Audits events received and queued as MMSIndications
#      MMS_LOG_RQUE - Audits queuing and matching of MMS Req/Conf pair
#
# ------- MMS Primitive Logging
#
#      MMS_LOG_REQ - Shows MMS Request parameters
#      MMS_LOG_IND -Shows MMS Request parameters received as anIndication
#      MMS_LOG_RESP - Shows MMS Response parameters
#      MMS_LOG_CONF-Shows MMSRequest parameters received as aConfirmation
#
# ------- Virtual Machine and MMS transaction object auditing
#
#      MMS_LOG_VM - currently not used
#      MMS_LOG_DATA -Logs data conversion to and from localrepresentation
#      MMS_LOG_PDU - Exposes MMS PDU's to the application
#
# ------- Reserved for Application Configuration Logging
#
#      MMS_LOG_CONFIG - reserved for customer or application use
#
# ------- Variable Access and Alternate Access Logging
#
#      MMS_LOG_RT - Logs creation of Runtime types
#      MMS_LOG_RTAA - Logs creation of Runtime types usingAlternateAccess
#      MMS_LOG_AA - Logs creation of AlternateAccess
#
# ------- SUIC Event Auditing
#
#      ACSE_IND_PRINT - Logs ACSE Indications
#      ACSE_CNF_PRINT - Logs ACSE Confirmations
#      ACSE_DEC_PRINT - Logs the initial parse of the ACSE PDU
#
# ------- DecNet/OSI Transaction Auditing
#
#      OSAK_PRINT - Shows DecNet OSI Suic transactions
#      OSAK_DEB - Shows DecNet OSI memory management
#
#      The remaining keywords control miscellaneous attributes of the
#      Log file entries or the location of the Log file.
#
#      Grammar errors that result from improper configuration are
#      reported in the file: mms_log.err.
#
#      Please refer to the Demo Configuration Appendix in the MMS-EASE
#      Reference Manual for further details.
#########################################################################
%LogBegin

# -------- General Logging Enable

  FileLogEn   # Log to a file instead of stdout
# MemLogEn    # Collect log entries in memory first
```

```
# -------- Misc. Control Flags
  TimeEnable= 2
  NoAppend    # means destroy existing file
  Setbuf      # use setbuf(fh,NULL) after open
# NoMsgheader # don't print message identifier
# NoWipe      # don't wipe bar in wrapped file
# NoWrap      # don't wrap the file
# HardFlush   # close and reopen the file each time

# ------- This combination is good for runtime use (errors only).
#
 MEM_LOG_ERR
 ASN1_LOG_ERR
 ASN1_LOG_NERR
 ACSE_ERR_PRINT
 MMS_LOG_ERR
 MMS_LOG_NERR
#
# ------- Combinations of Following are good for developing a program.
#
# MEM_LOG_CALLOC | MEM_LOG_MALLOC | MEM_LOG_REALLOC | MEM_LOG_FREE
#
# ASN1_LOG_DEC
# ASN1_LOG_ENC
#
  MMSCONF_PRINT
  MMSIND_PRINT
#
# MMS_LOG_DEC
# MMS_LOG_ENC
# MMS_LOG_ACSE
# MMS_LOG_LLC
# MMS_LOG_IQUE
# MMS_LOG_RQUE
# MMS_LOG_PDU
#
# MMS_LOG_REQ
# MMS_LOG_IND
# MMS_LOG_RESP
# MMS_LOG_CONF
#
# MMS_LOG_VM
# MMS_LOG_DATA
#
# MMS_LOG_CONFIG
#
# MMS_LOG_RT
# MMS_LOG_RTAA
# MMS_LOG_AA
# ACSE_IND_PRINT
# ACSE_CNF_PRINT
# ACSE_DEC_PRINT
#
# OSAK_PRINT
# OSAK_DEB
#
# -----This section can override the dynamic memory management defaults.
#       Contact SISCO Technical Support for proper use of memory
#       management options.
#
# ------- Internal Memory Management Auditing
#
# m_fill_en
# m_heap_check_enable
```

```
# m_check_list_enable
# m_find_node_enable
# m_track_prev_free
# list_debug_sel
# m_no_realloc_smaller

# -------  Log File Attributes

  LogFileSize= 500000
  LogFileName= mms.log

# -------  Memory Resident Logging Attributes for Timing Info

# LogMemItems= 1000
# DumpFileName= mmsdmem.log
# MemAutoDump

%LogEnd
```

IMPORTANT NOTE:

These Demo configuration files (**mmsdemo.cfg**, **mmsvar.cfg**, and **mms_log.cfg**) are only necessary for the Demonstration program supplied with MMS-EASE. Your application program may decide to implement configuration in a different manner. The Demo configuration files described here are not accessed in any way by the MMS-EASE Libraries.

MMSEASY Demonstration Program

In addition to the standard demo program, MMS-EASE products supply a simple console-based demo that shows how to connect to a remote device, to read a variable, and to disconnect. It can also be used in Server Mode which allows remote clients to connect and read certain variables.

This simple MMS-EASE application is contained all in one source module and is well commented. There is code for both a client and a server. As a client it simply reads a named variable from the specified server. As a server, it exposes two variables to the MMS network and allows client applications full read/write access to those variables. The server portion of this application also supports the GetNameList and GetVariableAccessAttributes services as a server. This code is intended only as a simple demonstration of what can be done easily, and does not make use of system specific event notification facilities provided by MMS-EASE.

Please consult your MMS-EASE installation guide for further instructions on how to build this demo program.