# MMS-EASE *Lite*
# Reference Manual

Revision 13

*Systems Integration Specialists Company, Inc.*
6605 19½ Mile Road, Sterling Heights, MI 48314-1408, USA
Tel: +586-254-0020, Fax: +586-254-0053
E-Mail: support@sisconet.com, URL: http:\\www.sisconet.com

Printed in U.S.A.

**08/2004** DWG#100275/2.3

# MMS-EASE *Lite*

## Reference Manual

## COPYRIGHT NOTICE

## DISCLAIMER

Systems Integration Specialists Company, Inc. makes no representation or warranties with respect to the contents of this manual and specifically disclaims any implied warranties of merchantability of fitness for any particular purpose. Further, Systems Integration Specialists Company, Inc. reserves the right to revise this publication and to make changes in it from time to time without obligation of Systems Integration Specialists Company, Inc. to notify any person or organization of such revision or changes.

All warranty and support for this product is provided by SISCO.

**Revision 13**

**08/12/04**

## Chapter 1

# Introduction

## What is MMS-EASE *Lite*?

SISCO's **MMS-EASE *Lite*** (**E**mbedded **A**pplication **S**ervice **E**lement) is a C language **A**pplication **P**rogram **I**nterface (API) for the **M**anufacturing **M**essage **S**pecification (MMS) protocol. It consists of source code modules derived from the MMS-EASE product line as well as a set of new files optimized for small system applications. These modules are compiler and operating system independent. MMS-EASE *Lite* has been created to minimize code and data space requirements and allows resource-limited devices to embed MMS within the device in a cost effective and resource efficient manner. It provides a mechanism for applications to encode and decode MMS PDUs. It shares the MMS-EASE data structures and a modified subset of the complete MMS-EASE API. In addition, there is an easy to use high-level application framework (MVL) designed to speed the development process. Contact SISCO for more information on available MMS-EASE *Lite* packages.

## About This Manual

The MMS-EASE *Lite* Reference Manual explains how to use MMS-EASE *Lite*. It explains how to encode and decode MMS PDUs. This manual is presented in nine sections:

- Chapter 1, Introduction, provides a brief overview of MMS-EASE *Lite*, and this document.
- Chapter 2, Getting Started, describes how to install and configure MMS-EASE *Lite*. It also describes how to use MMS-EASE *Lite* effectively.
- Chapter 3, Building MMS-EASE *Lite*, describes how to compile and link the MMS-EASE *Lite* libraries.
- Chapter 4, MMS-EASE *Lite* Lower Layers, describes the interaction of the MMS-EASE *Lite* Stack components.
- Chapter 5, MMS-EASE Lite Application Program Interfaces, describes the two interfaces of MMS-EASE *Lite*, the MVL and the MMS Protocol Encode/Decode interfaces.
- Chapter 6, Using MVL, documents MVL (**M**MS **V**irtual **L**ight). It includes an overview, object control structures and functions, as well as MVL Client and Server functionality.
- Chapter 7, Using MVL UCA Support, describes how to set up and use MVLU.
- Chapter 8, MMS Object Foundry, documents the MMS Object Foundry and its function.
- Chapter 9, IEC GOOSE and IEC GSSE Support, describes how to use IEC-61850 GOOSE and GSSE (IEC GSSE is the same as UCA GOOSE).

In addition, there are the following appendices:

- Appendix A, Subset Creation, provides steps on how to create applications that only use a subset of the supplied services.
- Appendix B, Logging Facilities, provides information regarding the **S**ISCO **Log**ging (**S_LOG**) system, a flexible and useful approach to system logging.

- Appendix C, Linked List Manipulation, documents the Linked List Manipulation functions that can be used in your application.
- Appendix D, Memory Management Tools, provides a set of memory management tools that include logging and integrity checking.
- Appendix E, GLBSEM Subsystem for Multi-threaded Support, addresses the issues related to writing a thread-safe MMS-EASE application.
- Appendix F, Support Functions, miscellaneous functions.
- Appendix G, Subnetwork API, describes the use of the Subnetwork layer and the rewriting of the API functions.
- Appendix H, MMS-EASE Type Description Language (TDL), provides information on TDL and includes several examples of how to build complex type definitions using the TDL.
- Appendix I, Logging for Lower Layers, provides information for diagnosing communication and other ACSE API related problems.
- Appendix J,  IEC GOOSE Example Application Framework, describes the sample application framework.

# Conventions used in this Manual

- Function names, structures, and members of functions and structures are shown in boldface **`Courier New`** type.
- Code fragments are shown in `Courier New`.
- File names are shown in **lowercase, bold Times New Roman**.

## Chapter 2

# Getting Started

## Prerequisites

Because of the technical nature of MMS-EASE *Lite*, and MMS, some level of knowledge is required by the user to fully understand how to use MMS-EASE *Lite*. You need to have familiarity with MMS specifications (particularly MMS: ISO IEC/IS 9506 and ISO DIS 9506). Information about the MMS specifications can be obtained from the following source:

SPECIFICATIONS:
ANSI (American National Standards Institute)
1430 Broadway
New York, NY  10018

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

In addition, if using IEC-61850, some level of knowledge is required by the user to fully understand how to use MMS-EASE *Lite*. You need to have familiarity with the following specifications:
IEC-61850 and UCA v 2.0 (IEEE-SA TR 1550-1999).

## Installation

The following installation procedures assume that you are familiar with your operating system and your computer.

***Note:*** *When installing software on a Windows machine, version information giving MMS-EASE Lite part number, location, and the major and minor version numbers are placed in the Windows registry. Also, a file called* ***mmsldefs.h*** *is found in the installation directory containing part number, version, and internal build number information. The definitions in this file may be used by the program, as shown in the sample programs provided. These two locations can be used to determine the version of MMS-EASE Lite installed on your system. Please refer to* ***HKEY_LOCAL_MACHINE\SOFTWARE\SISCO\MMS-EASE Lite\CurrentVersion*** *for related registry information.*

1. The product can be installed on Windows NT/2000/XP. If the files need to be moved to another computer, it is recommended that FTP be used to transfer the files after installation.
2. Insert the MMS-EASE *Lite* CD-ROM into the CD-ROM drive.

3. If the Autorun feature is enabled on your computer, go to Step 4. Otherwise, click on **Start**, select the **Run** option, and type the following command:

   ```
   {d}:\disk1\setup
   ```

   where {d} designates the letter of your CD-ROM drive.

4. When the MMS-EASE *Lite* setup initializes, you will be asked where to install the source code. The installation script will search the Windows Registry and try to find where the product was previously installed and install over the top of any existing installation. To install either MMS-LITE-801-001 or MMS-LITE-802-001, type in the Product Key as found on the label of your CD.
5. Follow the instructions on the screen to complete the MMS-EASE *Lite* installation.
6. If you need to install the OSI LLC Protocol Driver for Windows 2000, NT or XP, follow the instructions *Installing the OSI LLC Protocol Driver for Windows NT* on page 5 or *Installing the OSI LLC Protocol Driver for Windows 2000* on page 5 or *Installing the OSI LLC Protocol Driver for Windows XP* on page 6.

   **IMPORTANT:** If you are replacing an existing version of the OSI LLC Protocol Driver, you need to remove the current version. To remove the current version, follow the instructions under *Removing the Current Version of the OSI LLC Protocol Driver for Windows NT* on page 4 or *Removing the Current Version of the OSI LLC Protocol Driver for Windows 2000* on page 5.

7. Click on **Start**. From the **Settings** option, then select **Control Panel**. From the **Control Panel** Folder, select the **Network** shortcut.
8. Select the **Protocols** Dialog Tab and click the **Add...** button.
9. From the **Select Network Protocol** Dialog Box, click the **Have Disk...** button.
10. From the **Insert Disk** Dialog Box, type the path `{d}:\osillc\winnt`, where {d} designates the letter of your CD-ROM drive, and click on **OK** to accept the path.
11. From the **Select OEM Option** Dialog Box, click on **OK**.
12. Click the **Close** button and restart your computer.
13. The directory structure on the following pages should now exist on your computer.

# Removing the Current Version of the OSI LLC Protocol Driver for Windows NT

If you have a previous version of MMS-EASE *Lite*, it is recommended that the driver be replaced.

1. Click on **Start**. From the **Settings** option, select **Control Panel**.
2. From the **Control Panel** Folder, click on the **Network** shortcut.
3. Select the **Protocol** Dialog Tab, highlight the **OSI LLC Protocol Driver** and click the **Remove** button. Click on **OK**. In older versions of MMS-EASE *Lite*, select the **Services** tab to remove the protocol driver.
4. Click the **Close** button and reboot your computer.
5. When the system reboots, set directory to the Windows System Drivers (**System32\Drivers**) and delete the file **osillc.sys**. The installation process will fail to load the correct OSILLC device driver if an older version of the device driver is present in the system directory.
6. Restart your computer.

# Installing the OSI LLC Protocol Driver for Windows NT

To install the OSI LLC Protocol Driver for Windows NT, complete the following steps:

1. Click on the **Start** bar. From the **Settings** option, select **Control Panel**.
2. From the **Control Panel** Folder, select the **Network** shortcut.
3. Select the **Protocols** Dialog Tab and click the **Add...** button.
4. From the **Select Network Protocol** Dialog Box, click **Have Disk...**
5. From the **Insert Disk** Dialog Box, type the path **`{d}:\osillc\winnt`**, where {d} designates the letter of your CD-ROM drive, and click on **OK** to accept the path.
6. From the **Select OEM Option** Dialog Box, click on **OK**.
7. Click the **Close** button.
8. Restart your computer.

# Removing the Current Version of the OSI LLC Protocol Driver for Windows 2000

If you have a previous version of MMS-EASE *Lite*, it is recommended that the driver be replaced.

1. Click on the **Start** bar. From the **Settings** option, select **Control Panel**.
2. From the **Control Panel** Folder, select the **Network and Dial-up Connection** icon.
3. Double click on the **Local Area Connection** icon.
4. Click on the **Properties** button.
5. Highlight the **OSI LLC Protocol Driver** and click the **Uninstall** button.
6. Restart your computer.

# Installing the OSI LLC Protocol Driver for Windows 2000

To install the OSI LLC Protocol Driver for Windows 2000, complete the following:

1. Click on the **Start** bar. From the **Settings** option, select **Control Panel**.
2. From the **Control Panel** Folder, select the **Network and Dial-up Connection** icon.
3. Double click on the **Local Area Connection** icon.
4. Click on the **Properties** button.
5. Click on the **Install** button.
6. Select the network component type of **Protocol** and click on the **Add** button.
7. From the **Select Network Protocol** Dialog Box, click the **Have Disk...**
8. From the **Install from Disk** Dialog Box, type the path **`{d}:\osillc\win2000`**, where {d} designates the letter of your CD-ROM drive, and click on **OK** to accept the path.
9. From the **Select Network Protocol** Dialog Box, click on **OK**.
10. Click the **Close** button.
11. Restart your computer.

# Installing the OSI LLC Protocol Driver for Windows XP

Install the OSI LLC Protocol Driver for Windows XP by completing these steps:

1. Click on the **Start** bar. Select **Control Panel**.

   1.1 If you are in **Classic** view, got to step 2. If you are in the default **Category** View, read steps 1.2 and 1.3.

   1.2 Select the **Network and Internet Connections** icon.

   1.3 Go to Step 2.

2. Select the **Network Connections** icon.
3. Double click on the **Local Area Connection** icon.
4. Click on the **Properties** button.
5. Click on the **Install** button.
6. Select the network component type of **Protocol** and click on the **Add** button.
7. From the **Select Network Protocol** Dialog Box, click the **Have Disk...**
8. From the **Install from Disk** Dialog Box, type the path **{d}:\osillc\winxp**, where {d} designates the letter of your CD-ROM drive, or Click on the Browse button and double click on **osillc.inf** or select **osillc.inf** and click on the **Open** button.
9. Once the path to the **osillc.inf** file is entered, from the **Install From Disk** Dialog Box, click on **OK**.
10. From the Select **Network Protocol Dialog** Box select the **OSILLC Protocol Driver**
11. Click on the **OK** button
12. From the **Local Area Connection Properties** Dialog Box, click the **Close** button.
13. Restart your computer.

# Configuring the OSI LLC Protocol Driver for Windows NT/2000/XP

## *Configuring LSAPs on Windows NT/2000/XP*

To set in the registry the LSAPs that the driver will filter on, edit the registry key **HKEY_LOCAL_MACHINE\System\CurrentControlSet\Services\OSILLC\Parameters\LSAP**s by adding needed LSAPs.

FE - OSI Frames (default)

FB - UCA Time Synchronization Frames

This list may be extended or changed in the future.

## *Configuring Ethertype Packet Filtering on Windows*

To enable Ethertype packet filtering, follow the steps below:

1. Uninstall current OSILLC driver. This may require a system reboot.

2. Install version 2.15 of the OSILLC driver. This may require a system reboot.

At this point the driver will filter on all packets marked with the Virtual LAN ID (0x8100). To filter on specific Ethertype Ids complete the following steps:

1. Create a new Binary Value registry entry named 'EthertypeIDs' in the following path: **\HKEY_LOCAL_MACHINE\System\CurrentControlSet\Services\OSILLC\Parameters**. If this parameter does not exist when MMS-EASE *Lite* is installed, it is automatically created and the value is set to "88 b8 88 b9" which enables reception of Ethertype Ids 0x88b8 and 0x88b9 (these are the Ethertype Ids for IEC-61850 GOOSE and IEC-61850 GSE Management).
2. Set the Ethertype IDs you wish to filter on. For example, to filter on 0x8F1B, enter: 8F 1B. To filter on more than one ID, simply add them at the end of the list. For example to filter on 0x8F1B and 0x8000 enter: 8F 1B 80 00
3. Reboot for your changes to take affect.

## *Directory Structure*

| | |
|---|---|
| \mmslite | (root MMS-EASE *Lite* directory) |
|  \cmd | (command files for creating binaries) |
|   \gnu | (GNU makefiles for Linux,  QNX, etc.) |
|   \pharlap | (Pharlap makefiles) |
|   \win32 | (Win32 project files) |
|  \src | (source code – all .c files) |
|  \inc | include files – all .h files) |
|   \mmsop_en | (default **mmsop_en.h** file) |
|  \uca | (stack profiles source root) |
|   \acse | (ACSE source) |
|    \sm_test | (Serial Manager tools) |
|    \sm_targt | (serial manager server) |
|    \sm_test | (serial manager client) |
|   \goose | (GOOSE source)* |
|   \leant | (TP4, TP0, CLNP, ES-IS, subnet, UCA time sync source) |
|    \sn_test | (subnetwork test tools) |
|    \sn_targt | (subnet server) |
|    \sn_test | (subnet client) |
|   \rs | (reduced stack source)* |
|  \bin | (utility executables) |
|  \mvl | (MMS Virtual Light) |
|   \src | (MVL source) |
|    \acse | (MVL-ACSE source) |
|    \loop | (loopback LLP files) |
|   \usr | (MVL sample user root) |
|    \client | (MVL sample client) |
|    \server | (MVL sample server) |
|    \uca_srvr | (UCA sample server) |
|    \iecgoose | (IEC GOOSE framework sample) |
|    \scl_srvr | (IEC_61850 sample server using SCL) |
|   \util | (MVL utility root) |
|    \foundry | (MVL foundry) |
|     \linux | (contains structure alignment configuration file for Linux) |
|     \qnx | (contains structure alignment configuration file for QNX) |
|     \win32 | (Win32 makefiles) |
|     \uca09 | (GOMSFE Rev 9 UCA model files) |
|    \mbufcalc | MVL buffer init support) |
|    \mmslog | (MMS PDU decoder/analyzer) |
|   \Gsemtest | (Global semaphore test code) |
|  \obj | (object code) |
|  \doc | (PDF documentation) |
|  \lib | (libraries) |

* Exists only for OSI Installations

\mmslite (cont'd)

| | |
|---|---|
| \win32lib | (Win 32 libraries) |
| \osillc | (OSILLC driver source code) |
| \osillc95 | (Windows 98 source files) |
| \osillccommon | (OSILLC common files) |
| \osillcnt | (Windows NT/2000/XP source files) |

## Chapter 3

# Building MMS-EASE *Lite*

Many embedded environments require the use of a cross compiler which runs on a “host” computer and the resulting programs are transferred to the “target” system for execution. In this discussion, the term “host” will refer to the environment where the application is compiled and linked, and “target” will refer to the environment where the application is to be executed. See the following chapter for more details.

The steps below are all to be executed on the host system, and will result in a set of libraries that can be used to create a MMS-EASE *Lite* application that can be transferred to and executed on the target system.

1. Edit **glbtypes.h** and map the SISCO data types onto the target system’s native C data types. This file contains type definitions for many sample environments.
2. Edit **sysincs.h** and select appropriate system header files for the development environment.
3. Review system specific code, such as data alignment, floating-point format, and high-resolution timers.
4. Review and port stack components. See page 31 for information on portation.
5. Modify the MMS-EASE *Lite* make files as required to allow building target libraries on the host.
6. Build the MMS-EASE *Lite* libraries to be used in creating applications for the target environment.
7. Edit the file **align.cfg** in order to specify the alignment requirements of the target environment. Samples are included for QNX and WIN32 in subdirectories under **\mmslite\mvl\util\foundry**. If the target system alignment requirements are not well known, compile and execute the executable **findalgn.exe** in the target environment.

### *Global Variable Initialization*

MMS-EASE *Lite* has many global variables, some of which are initialized at compile time and which may be changed during program execution. This can cause problems in some environments where initialized global data is placed in a code segment and is subject to checksum verification. The define, **`NO_GLB_VAR_INIT`**, can be used in the source code to avoid global variable initializations. If this feature is used, the user application must call **`mvl_init_glb_vars`** before any other MVL or MMS-EASE *Lite* activity.

## mvl_init_glb_vars

**Usage:** This function initializes all global variables that can not be initialized by the compiler.

**Function Prototype:** `ST_VOID mvl_init_glb_vars (ST_VOID);`

**Parameters:** NONE

**Return Value:** `ST_VOID`

# Development System Preparation

The following items need to be considered before the libraries can be created for MMS-EASE *Lite*.

## Conditional Compilation Defines

MMS-EASE *Lite* is a flexible code base that can be used to create a variety of communications profiles. This is accomplished by a set of defines that are used for conditional compilation of profile specific code. The defines used for all this purpose are listed below. Note that this list does not contain those defines that are used to establish capacities.

| **PRIMARY GENERAL DEFINES** | **DESCRIPTION** |
|---|---|
| `MMS_LITE` | This define is required in order to compile the MMS-EASE *Lite* MMS source code.<br>This define is used in the standard product makefiles. |
| `DEBUG_SISCO` | This define is used in most SISCO software components and has two purposes:<br>1. Compile in logging statements. This adds significantly to the size of the static string space, and slows things down just a bit. SISCO recommends using this define where possible.<br>2. Compile in some level of debug error level checking. This is a secondary effect of using the `DEBUG_SISCO` define.<br>3. Compile in memory allocation debug calls. These are used to track the file/line number of all allocations<br>This define is used in the standard product makefiles in creating the debug libraries. |
| `NO_GLB_VAR_INIT` | This define is used when target environments do not support compile time data initializations, which otherwise are used in several places within MMS-EASE *Lite*. This is NOT defined in the standard product makefiles. |
| `S_MT_SUPPORT` | This define is used to enable multithread support in the various MMS-EASE *Lite* libraries. This is NOT defined in the standard product makefiles except for Windows. |
| `SD_BYTE_ORDER` | This must be defined in **glbtypes.h** for each platform to indicate the byte order used to store data (big-endian or little-endian). It must be set to `SD_BIG_ENDIAN` if the platform is “big-endian”. It must be set to `SD_LITTLE_ENDIAN` if the platform is “little-endian”. |
| `UNICODE_LOCAL_FORMAT` | This define slects the local format used to store Unicode strings. According to the MMS Specification, Unicode strings must always be encoded in UTF8 format. This is also the format that most UNIX systems use to store Unicode strings. However, some systems (e.g., Windows) store Unicode strings in UTF16 format. The ASN.1 encoder converts Unicode strings from the local format to UTF8. The ASN.1 decoder converts from UTF8 to the local format. This conversion is controlled by this define. It must be defined as |

| | |
|---|---|
| | UNICODE_UTF8 or UNICODE_UTF16. It is currently defined in **asn1r.h** to be UNICODE_UTF8 for all systems except Windows as follows:<br>`#if !defined(UNICODE_LOCAL_FORMAT)`<br>`#if defined(_WIN32)`<br>`#define UNICODE_LOCAL_FORMAT UNICODE_UTF16`<br>`#else`<br>`#define UNICODE_LOCAL_FORMAT UNICODE_UTF8`<br>`/*default format   */`<br>`#endif`<br>`#endif` |
| **PRIMARY MVL DEFINES** | |
| MVL_UCA | This define enables the UCA support in MVL. This is used when compiling the mvlu library. |
| MVL_AA_SUPP | This define allows MVL to support alternate access as a variable access server. This is normally defined in **mvl_defs.h** but may be undefined if the user application does not need alternate access support and memory constraints dictate minimum possible size. |
| MVL_DYN_MEM | This define is used to allow MVL to make use of dynamic memory allocation as required. The alternative is that MVL will allocate a set of buffers at initialization time and will keep them for the life of the application. This is normally defined in **mvl_defs.h**, and must be defined for server applications using the MVL asynchronous response mechanism. |
| MVL_DYN_ASN1_TYPES | This define is used to allow MVL to generate the ASN.1 type from the Runtime Type as required (typically for GetVariableAccessAttributes as a server). The alternative is to have the ASN.1 encoded type specifications attached to the MVL type control, which may be more costly in terms of memory usage, especially for large number of types. This is normally defined in **mvl_defs.h**. |
| MVL_INFO_RPT_CLIENT | This define must be used for client applications that will be receiving information reports. For other applications, it will simply increase the size of the Variable Association data structure for no good reason. This is normally defined in **mvl_defs.h**, and may be commented out for server only applications. |
| ICCP_LITE_SUPP | This define is found in **mvl_defs.h** to expose members in MVL structures for use by MMS-EASE *Lite* with TASE.2 Extensions. Comment this define out to build a MVL library optimized for space that is not to be used for ICCP. This define is uncommented by default. |
| ICCP_LITE | This define must be used by all applications using MMS-EASE *Lite* with TASE.2 Extensions. It causes ICCP specific pieces of code to be compiled into **mvl_acse.c**. |
| **SECONDARY MVL DEFINES** | |
| ALLOW_MULTIPLE_REQUESTS_OUT | This define is used to allow MVL client applications to have more than one request outstanding but requires the **gen_list** list management services. It is the best choice unless memory constraints require absolute minimum size. |

| | |
|---|---|
| | This is normally defined in **mvl_serv.c.** |
| NEGIOTIATE_INITIATE_PARAM | This define allows **mvl_acse.c** to negotiate the parameters used to send an initiate response, based on the supplied initiate response parameters. This is normally defined in **mvl_acse.c**, but may be undefined if the user application performs the negotiation process itself. |
| CLACSE | This define allows MVL to make use of connectionless ACSE services. This is normally not defined. |
| MVL_DESCR_SUPP | This define is used to compile in support for Described variable access. This feature is not fully implemented in MMS-EASE *Lite* V4.xx. |
| MVL_XNAME | Compiling the MVL library with this define causes the *xName* member to be exposed in **MVLU_RD_VA_CTRL**, and **MVLU_WR_VA_CTRL** typedefs. This define allows the fully qualified UCA variable name to be passed in to UCA variable read and write functions. By default, **MVL_XNAME** is commented out in **mvl_defs.h** and the feature is not enabled. |
| USE_RT_TYPE_2 | This define allows named components to be added to dynamically created types. It is possible to use both MMS Object Foundry and dynamically created types in an application when **USE_RT_TYPE_2** is defined. By default, **USE_RT_TYPE_2** is commented out in **mms_vvar.h**. Please refer to the function **mvl_type_id_create**. |
| **PRIMARY NETWORK STACK DEFINES** | |
| MOSI | This define is used to select the minimal OSI profile when compiling the ACSE & Lean-T software modules. The **LEAN_T** define must also be defined when using the MOSI define. |
| LEAN_T | This define is used to enable Transport layer code. It is required if OSI or TCP/IP layers are included in the stack. |

| | |
|---|---|
| REDUCED_STACK | This define is used to select the UCA Reduced Stack profile when compiling the ACSE, MVL, and sample application software modules. |
| UCA_SMP | This define is used when compiling UCA network layer and sample application modules to enable use of the UCA Time Synchronization protocol. This is defined in the standard product makefiles. |
| **SECONDARY NETWORK STACK DEFINES** | |
| CALLED_ONLY | This define can be used to reduce the size of the application when it will not initiate connections. |
| CALLING_ONLY | This define can be used to reduce the size of the application when it will not receive initiate indications. |
| CLNP_STAT | This define allows CLNP to record statistics. |
| TP0_ENABLED | This define is used to enable TP0 functionality in the Lean-T software modules. This is required for TCP/IP (via RFC1006) support. |
| TP4_ENABLED | This define is used to enable TP4 functionality in the Lean-T software modules. This is required for OSI support. |
| **PRIMARY MMS DEFINES** | |
| BTOD_DATA_SUPPORT | This define is used to enable support for binary time of day data types This is defined in the standard product makefiles. |
| TIME_DATA_SUPPORT | This define is used to enable support for generalized time data types. This is defined in the standard product makefiles. |
| FLOAT_DATA_SUPPORT | This define is used to enable support for floating point data types. This is defined in the standard product makefiles. |
| INT64_SUPPORT | This define is used to enable support for 64 bit integer data types This is defined in the standard product makefiles for WIN32, and is not defined for DOS. This needs to be examined when porting to other platforms. |
| **SECONDARY MMS DEFINES** | |
| CS_SUPPORT | This define is used to enable support for MMS companion standards. This is NOT defined in the standard product makefiles, and is not supported in MVL in any way. |
| MOD_SUPPORT | This define is used to enable support for MMS modifiers. This is NOT defined in the standard product makefiles, and is not supported in MVL in any way. |
| ASN1_ARB_FLOAT | This define is used to enable MMS to decode all forms of floating point data. When it is not defined, only IEEE 754 format floating point data can be decoded. This is defined by default. |
| GET_CONSTRUCTED_BSTRINGS | This define is used to compile in ASN.1 code for decoding constructed bitstrings. This is not normally required, and is NOT defined in the standard product makefiles. |
| USE_COMPACT_MMS_STRUCTS | This define controls the makeup of some MMS-EASE data |

| | |
|---|---|
| | structures and allows a more compact form to be used. This is defined when **`MMS_LITE`** is defined. |
| **SAMPLE MVL APPLICATION DEFINES** | |
| `USE_MANUFACTURED_OBJS` | This define is used in **server.c** to compile in code related to using manufactured variables and variable lists. This is done to clearly show the mechanisms required. |
| `HARD_CODED_CFG` | This define is used in **server.c** to compile in code related to using hard coded configuration information instead of configuration files. By default, this is not defined. |
| `USE_FRAMEWORK_THREADS` | This define is only used in the IEC GOOSE Framework application (in iecgoose directory). It enables multi-threading code. |
| **OTHER DEFINES** | |
| `NO_REALLOC_SMALLER` | This define can be used when compiling the memory allocation tools to not **`realloc`** when the new size is less than the old. This can sometimes be helpful in reducing memory fragmentation. This is NOT defined in the standard product makefiles. |
| `USE_PCHRT` | This define can be used when compiling stime.c to enable use of the Ryle Design PC Timer Tools high-resolution timer libraries. This is not defined in the standard product makefiles. |
| `MEM_FILL_CONTROL` | This define can be used when compiling the memory allocation tools to have **mem_chks.c** overwrite the control header as well as the body of the buffer being freed. This is not defined in the standard product makefiles. |
| `MLOG_ENABLE` | This define can be used when compiling **mmsop_en.h** to create the function pointer tables for the MLOG subsystem. By default this is not defined when **`MMS_LITE`** is defined. |

## glbtypes.h

To promote portability and reduce name space conflicts, SISCO makes use of a set of defines in place of C data types. For instance, SISCO code uses **`ST_INT`** in place of the standard “int” keyword. Many of the defines are used to select data types with known precision. These defines can be found in the header file **glbtypes.h**. This file contains the defines for many operating systems and compilers, but it may be necessary to add a section for the target development environment.

This file defines the typedefs for the basic types used by MMS-EASE *Lite* for the system, along with some common defines and operating system and hardware select defines. A starting point for a new portation is the **`NEW_SYSTEM`**” **`SYSTEM_SEL`** value.

This can be used in one of two ways:

1. If the source code is going to be used to generate libraries for a single system, simply edit the **`NEW_SYSTEM`** section and modify the **`NEW_SYSTEM_DETECT`** define to one that uniquely identifies the compiler, then review the use of the two defines there. The **`NEW_SYSTEM`** define is used in the commonly required places in the MMS-EASE source code, so this may be all that is required.
2. If the source code is going to be used to generate libraries for multiple systems, it will be necessary to add the systems to the bit masked **`SYSTEM_SEL`** section for each of the target systems. To do this, simply use the next unused bit and assign a name to the value, then copy the **`NEW_SYSTEM`** section and modify to suit. Possible modifications are listed below:

    `SYSTEM_SEL` This is a bit masked define used with the logical OR to select appropriate system specific code. With MMS-EASE *Lite* in particular, this define is used in the ASN.1 tools for floating point conversions.

    Typically this can be left as is, as it selects the generic **`NEW_SYSTEM`** system as the target, and unless the code is to support multiple targets it should not be necessary to add another.

    `END_STRUCT` This define is included at the end of all MMS-EASE data structures that may be embedded within or “attached” to other MMS-EASE data structures. This typically is defined to be nothing, but for some hardware/compiler combinations (especially 64 bit RISC systems) may need to be defined to **`ULONG end_of;`** to force alignment on a quadword boundary.

# Unicode porting issues

If the default local format is not correct for your platform, the **UNICODE_LOCAL_FORMAT** define must be changed. In this case, it should be defined before including **asn1r.h**, preferably in **glbopt.h**). On systems with a local format of **UNICODE_UTF16**, the functions described below must be ported. They are already ported for Windows.

## asn1r_utf8_to_local

**Usage:** On systems where the local format is UTF16, this functions converts from a UTF8 string to UTF16 string. The destination string (**dst_ptr**) does NOT need to be NULL terminated (the calling function does that).

**Function Prototype:**

```
ST_INT asn1r_utf8_to_local (ST_CHAR *dst_ptr,
                            ST_INT dst_len,
                            ST_CHAR *src_ptr,
                            ST_INT src_len);
```

**Parameters:**

| | |
|---|---|
| dst_ptr | A pointer to the destination UTF16 string. |
| dst_len | The number of bytes in the destination UTF16 string. |
| src_ptr | A pointer to the source UTF8 string. |
| src_len | The number of bytes in the source UTF8 string. |

**Return Value:** Returns the number of bytes in the destination UTF16 string (**dest_ptr**) (may include the **NULL** terminator).

### asn1r_local_to_utf8

**Usage:** On systems where the local format is UTF16, this function converts from UTF16 string to UTF8 string. The source string (**`src_ptr`**) must be NULL terminated.

**Function Prototype:**

```
ST_INT asn1r_local_to_utf8 (ST_CHAR *dst_ptr,
                            ST_INT dst_len,
                            ST_CHAR *src_ptr);
```

**Parameters:**

| | |
|---|---|
| `dst_ptr` | A pointer to the destination UTF8 string. |
| `dst_len` | The number of bytes available in the destination UTF8 string. |
| `src_ptr` | A pointer to the source UTF16 string. |

**Return Value:** Returns the number of bytes in the destination UTF8 string (**`dst_ptr`**) (not including the **`NULL`** terminator).

## sysincs.h

This file is used within MMS-EASE *Lite* to select the system include files to be included. Review the section for **`SYSTEM_SEL == NEW_SYSTEM`** to verify that the target compiler supports the specified include files, and modify as required.

## Floating Point Representation

The ASN.1 floating point handling routines will need to be reviewed. These functions can be found in **ae_float.c** and **ad_float.c**. Note that the user sample application, **var.c,** can be used to verify that the floating point conversions are correct - in many cases no system specific work will need to be done here. IEEE 754 format is supported with no changes.

## Data Alignment

MMS-EASE *Lite* is designed to be able to present arbitrary data types in local C format for ease of use by the application programmer. As different compilers perform different "padding" in data structures, it may be necessary to review and/or create an appropriate data alignment table. This data alignment table will be used in the **align.cfg** file, and in the source module **mms_tdef.c**.

## High Resolution Timers

Some of the UCA profile components make use of high-resolution timer functions. The required resolution depends on the application (e.g., MAS Radios) but it is desirable to achieve  < 10ms resolution if possible. The source module to be examined is **stime.c**.

## Memory Allocation

MMS-EASE *Lite* allocates memory as required using an intermediate layer that is referred to as the **mem_chk** library. This library makes use of the standard malloc family of calls to execute the memory allocation/free requests, and optionally provides significant debugging assistance such as invalid free, buffer overwrites, and usage tracking. Note that these calls may be customized as required for the target system. The primary allocation functions are **`chk_malloc`**, **`chk_calloc`**, **`chk_realloc`**, **`chk_strdup`**, and **`chk_free`**. MMS-EASE *Lite* contains both full and Lite versions of the **mem_chk** libraries (the "Lite" version is **mem_chkl.c**).

MMS-EASE *Lite* treats memory allocation failures as fatal, non-recoverable errors. The user can elect to be notified via function pointer when memory allocation failures are detected, and can return valid **`malloc`** memory or can not return. The sample applications demonstrate use of these features.

## Logging Mechanisms

If possible, all developers to implement a logging subsystem in the target application. All SISCO components can perform error and debug logging that is controlled by bit-masked control variables. This logging code can be compiled in by using the define **`DEBUG_SISCO`**. The logging subsystem used by MMS-EASE *Lite* is called "slog" (**S**ISCO **Log**ging), and there are two versions supplied. The full-featured SLOG library is included. This library provides selectable file, memory, and user defined log streams. A "Lite" version of SLOG (SLOGL) is included. The source for this library is presented in stub form, to be customized for the target environment; for instance, it may be modified to log to a serial port or some other mechanism specific to the implementation.

The MVL sample applications demonstrate the use of the logging subsystem. Note that these samples make use of SLOG for application level logging as well as MMS-EASE *Lite* internal logging.

## *SLOG Feature Summary*

Below is a list of some of the features provided by the **`S_LOG`** system.

**GENERAL LOGGING**

- 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`** has the capability of using multiple logging control elements with one log file per logging control element.
- Includes Source file and Line Number information for debugging.
- In-memory logging is available for profiling and high performance applications.

**FILE LOGGING**

- **`S_LOG`** logs to circular file.
- It allows dynamic enabling and disabling of file logging using the supplied functions.
- Some controllable options include the following:

| | |
|---|---|
| **File Name** | The name of the log file can be indicated. |
| **File Size** | The size of the file can be changed. The size of this file is defaulted to 1M. |
| **Wipe Bar** | This is a set of comment lines indicating where new logging has started. |
| **File Wrap** | The data in the log file can be wrapped to make the file circular. |
| **Message Header** | A message identifier can be printed. |

**MEMORY LOGGING**

- **`S_LOG`** can log 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.

## Global Variables

The following global variables are used by MMS-EASE *Lite*.

```
MVL_CFG_INFO *mvl_cfg_info;         /* critical config parameters */
ST_UCHAR *mmsl_enc_buf;             /* encode buffer              */
MVL_NET_INFO mvl_calling_conn_ctrl; /* array of calling connection*/
                                    /* control structures         */
MVL_NET_INFO *mvl_called_conn_ctrl; /* array of called connection */
                                    /* control structures         */
```

# Creating MMS-EASE *Lite* Libraries

As MMS-EASE *Lite* is provided in source code form only, the first step is to create the object libraries required. When executed correctly, this procedure will result in the following libraries. Note that there are four configurations provided for most libraries. The debug versions contain additional code for logging and error checking. Further, note that not all libraries are to be created for all supported build environments.

There are four configurations for each project: “Release No Logging”, “Release Logging”, “Debug No Logging”, and “Debug Logging”. These configurations will exist for all projects: libraries, samples, and utilities, and a suffix convention is used to identify the configuration. The table below summarizes the configurations.

| Configuration | Debug | SLOG | Suffix _x | Comment |
|---|---|---|---|---|
| Release/Logging | No debug | Yes | “_l”<br>example: mmsl_l.lib | This is the configuration typically used for both development and deployment of applications. It supports **S**ISCO **log**ging (SLOG), but has no debugging information. |
| Release/No Logging | No debug | No | “_n”<br>example: mmsl_n.lib | This configuration can be used for deployment when the application is not to make use of SISCO logging. This may be useful for PharLap developers. |
| Debug/Logging | C7 compatible | Yes | “_ld”<br>example: mmsl_ld.lib | This configuration is used when debugging problems within the MMS-EASE Lite components. It can also be useful for use in field diagnostics. |
| Debug /No Logging | C7 compatible | No | “_nd”<br>example: mmsl_nd.lib | This configuration is used when debugging problems within the MMS-EASE Lite components without SISCO logging. This may be useful for PharLap developers. |

The following are libraries provided by MMS-EASE *Lite* where “_x” is one of the suffixes listed in the above table:

| | |
|---|---|
| **asn1_x.lib** | ASN.1 encode/decode libraries |
| **mem_x.lib** | Full featured memory management library |
| **meml_x.lib** | “Lite” memory allocation facility (template) |
| **mlogl_x.lib** | MMS-EASE *Lite* MMS service logging subsystem libraries |
| **mmsl_x.lib** | MMS PDU encode/decode libraries |
| **mmsle_x.lib** | Extended MMS PDU encode/decode libraries |
| **mvl_x.lib** | MMS Virtual Light (MVL) application framework libraries |
| **slog_x.lib** | Full file system based logging libraries |
| **slogl_x.lib** | “Lite” logging libraries (template) |
| **util_x.lib** | MMS-EASE *Lite* Utility libraries |

# Windows Batch Build

There are three build driver projects to the MMS-EASE *Lite* distribution and workspace for Windows systems; MakeLibs, MakeUtils, and MakeSamples. These projects create a null application, but have dependencies that allow all libraries, utilities, or sample applications to be created easily. The process used to build all elements in all configurations is as follows:

1. Set the MakeLibs as active project

    1.1. Select and build the ‘Release No Logging’ configuration

    1.2. Select and build the ‘Release’ configuration

    1.3. Select and build the ‘Debug No Logging’ configuration

    1.4. Select and build the ‘Debug’ configuration

2. Set the MakeUtils as active project

    2.1. Select and build the ‘Release No Logging’ configuration

    2.2. Select and build the ‘Release’ configuration

    2.3. Select and build the ‘Debug No Logging’ configuration

    2.4. Select and build the ‘Debug’ configuration

3. Set the MakeSamples as active project

    3.1. Select and build the ‘Release No Logging’ configuration

    3.2. Select and build the ‘Release’ configuration

    3.3. Select and build the ‘Debug No Logging ’ configuration

    3.4. Select and build the ‘Debug’ configuration

## User Migration Issues

Developers will have to select different SISCO libraries for their project, as the old names are no longer used. On Windows, projects have been modified to use “dependencies” so that library names do not have to be included in the link command. It is highly recommended that customer’s projects use this feature.

## WIN32 Development Environment

MMS-EASE *Lite* includes workspace and project files for Microsoft Visual Studio V6.0, and these files are located in **\mmslite\cmd\win32**. The following WIN32 projects are included in the main Microsoft Developer Studio workspace, which is **mmslite.dsw**. The projects can be built in batch mode or individually, but should be built in the following order:

1. Libraries
2. Utility applications
3. Sample applications

| | |
|---|---|
| makelibs.dsp | Makes all the libraries |
| makesamples.dsp | Makes all the samples |
| makeutils.dsp | Makes all the utilities |
| | |
| asn1.dsp | ASN.1 encode/decode library |
| mem.dsp | Memory allocation library - full featured version |
| meml.dsp | Memory allocation library - *Lite* version |
| mlog.dsp | MMS operation specific logging library |
| mmsl.dsp | Main MMS encode/decode library |
| mmsle.dsp | Extended MMS encode/decode library |
| mmslog.dsp | MMS logging library |
| mvl.dsp | MVL library |
| mvlu.dsp | MVL UCA library |
| ositcpe.dsp | TCP/IP (via RFC1006) stack library |
| ositcps.dsp | TCP/IP (via RFC1006) stack library using non-blocking sockets |
| ositp4e.dsp | 7 Layer OSI over Ethernet library |
| ositpxe.dsp | Library that includes TCP/IP (via RFC1006) and 7 Layer OSI over Ethernet |
| ositpxs.dsp | Library that includes TCP/IP (via RFC1006) using non-blocking sockets and 7 Layer OSI over Ethernet |
| ssec0.dsp | Required library for compatibility with future enhancements. |
| slog.dsp | SISCO logging library - full featured version |
| slogl.dsp | SISCO logging library - *Lite* version |

| | |
|---|---|
| smem.dsp | Memory allocation library using “pools”. |
| util.dsp | SISCO utility library |
| **Utility applications** | |
| foundry.dsp | **foundry.exe** utility application |
| mbufcalc.dsp | **mbufcalc.exe** utility application (obsolete) |
| iecgoose.dsp | IEC GOOSE Framework sample application |
| **Sample Applications** | |
| cositcpe.dsp | Client sample application for TCP/IP (via RFC1006) |
| cositcps0.dsp | Client sample application for TCP/IP using **ositcps** stack library |
| cositp4e.dsp | Client sample application for 7 Layer OSI over Ethernet |
| cositpxe.dsp | Client sample application for TCP/IP and 7 Layer OSI over Ethernet |
| cositpxs0.dsp | Client sample application for TCP/IP and 7 Layer OSI over Ethernet using **ositpxs** stack library |
| scl_srvr.dsp | IEC-61850 Server sample application using SCL |
| sositcpe.dsp | Server sample application for TCP/IP (via RFC1006) |
| sositcps0.dsp | Server sample application for TCP/IP using **ositcps** stack library |
| sositp4e.dsp | Server sample application for 7 Layer OSI over Ethernet |
| sositpxe.dsp | Server sample application for TCP/IP and 7 Layer OSI over Ethernet |
| sositpxs0.dsp | Server sample application for TCP/IP and 7 Layer OSI over Ethernet using **ositpxs** stack library |
| uositcpe.dsp | UCA Server sample application for TCP/IP (via RFC 1006) |
| uositcps0.dsp | UCA Server sample application for TCP/IP using **ositcps** stack library |
| uositp4e.dsp | UCA Server sample application for 7 Layer OSI over Ethernet |
| uositpxs0.dsp | UCA Server sample application for TCP/IP and 7 Layer OSI over Ethernet using **ositpxs** stack library |
| uositpxe.dsp | UCA Server sample application for TCP/IP and 7 Layer OSI over Ethernet |

# GNU Development Environment

MMS-EASE *Lite* includes makefiles that work with the GNU Make utility that is available on many UNIX-like platforms. These files are located in **\mmslite\cmd\gnu**. These makefiles should work with little or no modification on any system using GNU Make or a similar UNIX-like make utility. A shell script **`mkall.sh`** is provided to execute all the necessary make commands and to build everything in the following order:

1. Libraries
2. Utility applications
3. Sample applications

| | |
|---|---|
| asn1.mak | ASN.1 encode/decode library |

| | |
|---|---|
| client.mak | Client sample application for TCP/IP using **ositcpe** stack library |
| cositcps0.mak | Client sample application for TCP/IP using **ositcps** stack library |
| findalgn.mak | **findalgn.exe** utility application |
| foundry.mak | **foundry.exe** utility application |
| mbufcalc.mak | **mbufcalc.exe** utility application (obsolete) |
| mem.mak | Memory allocation library - full featured version |
| meml.mak | Memory allocation library - *Lite* version |
| mlogl.mak | MMS operation specific logging library - *Lite* version |
| mmsl.mak | Main MMS encode/decode library |
| mmsle.mak | Extended MMS encode/decode library |
| mmslog.mak | MMS logging library |
| mvl.mak | MVL library |
| mvlu.mak | MVL UCA library |
| ositcpe.mak | TCP/IP (via RFC1006) stack library |
| ositcps.mak | TCP/IP (via RFC1006) stack library using non-blocking sockets |
| scl_srvr.mak | IEC-61850 Server sample application using SCL |
| server.mak | Sample Server application for TCP/UP using **ositcpe** library |
| slistend.mak | TCP/IP socket listen task |
| slog.mak | SISCO logging library - full featured version |
| smem.mak | Memory allocation library using “pools”. |
| sositcp0.mak | Server sample application for TCP/IP using **ositcps** library |
| sreadd.mak | TCP/IP socket read task |
| ssec0.mak | Required library for compatibility with future enhancements. |
| uca_srvr.mak | IEC-61850/UCA Server sample application for TCP/IP using **ositcpe** lib |
| uositcps0.mak | IEC-61850/UCA Server sample application for TCP/IP using **ositcps** lib |
| util.mak | SISCO utility library |

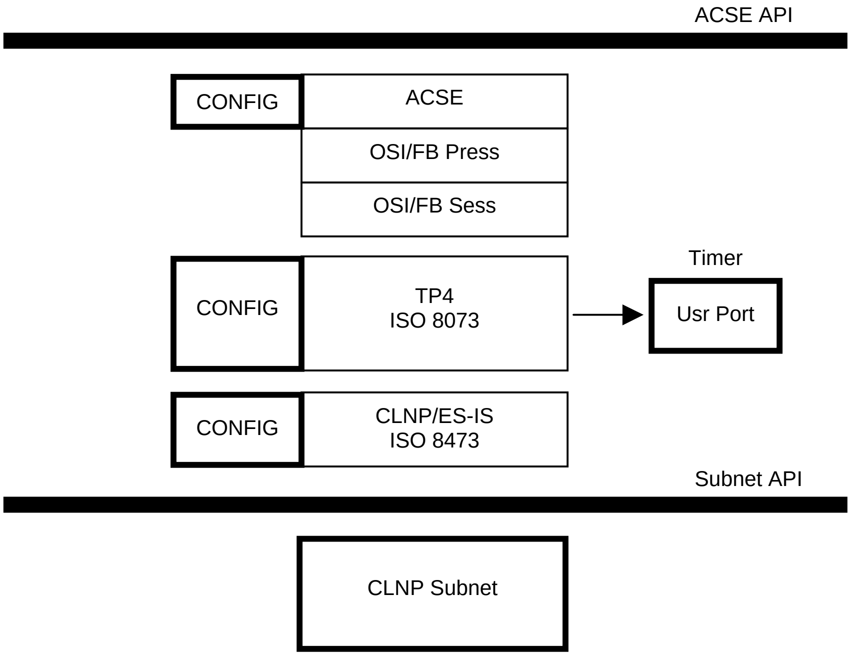
Chapter 4

# MMS-EASE *Lite* Lower Layers

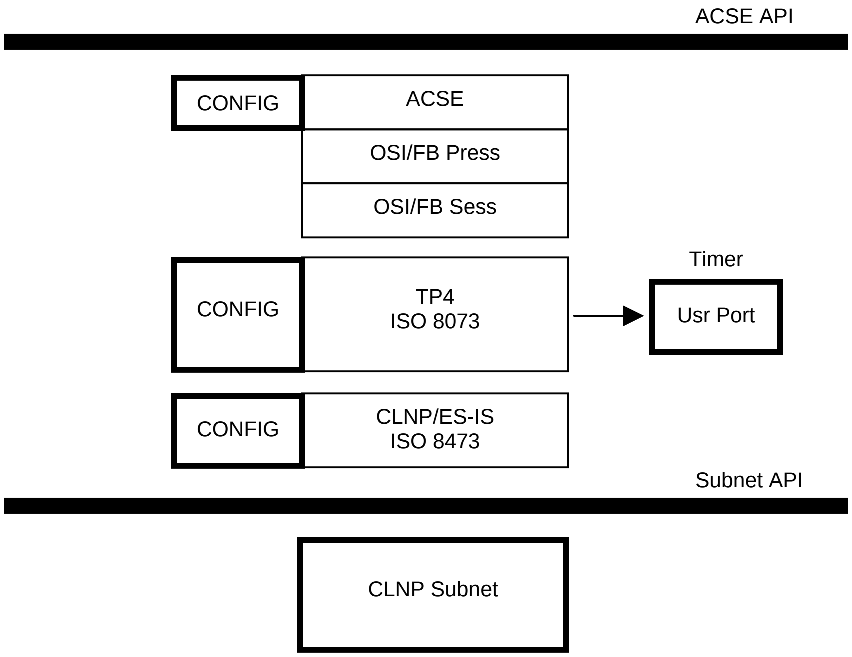
## Profile Options

The “MMS-EASE *Lite* Stack Components” is an implementation of various Open Systems Interconnection (OSI) protocol layers. It is designed for systems with very limited resources such as some embedded systems and to be modular so that only the protocol layers required for a particular application need to be used. It consists of several C source code modules which can easily be compiled for any embedded system. It contains only ANSI standard C except for a few simple functions (isolated in the **tp4port.c** module) which may need to be modified for a particular system. In the terminology of the OSI Reference Model, each protocol layer is described as providing “services” to the layer above it. In this implementation, these “services” are provided by means of an Application Programming Interface (API) which is simply a C function call interface. The diagrams below show the relationships between the OSI protocol layers and the APIs between them.

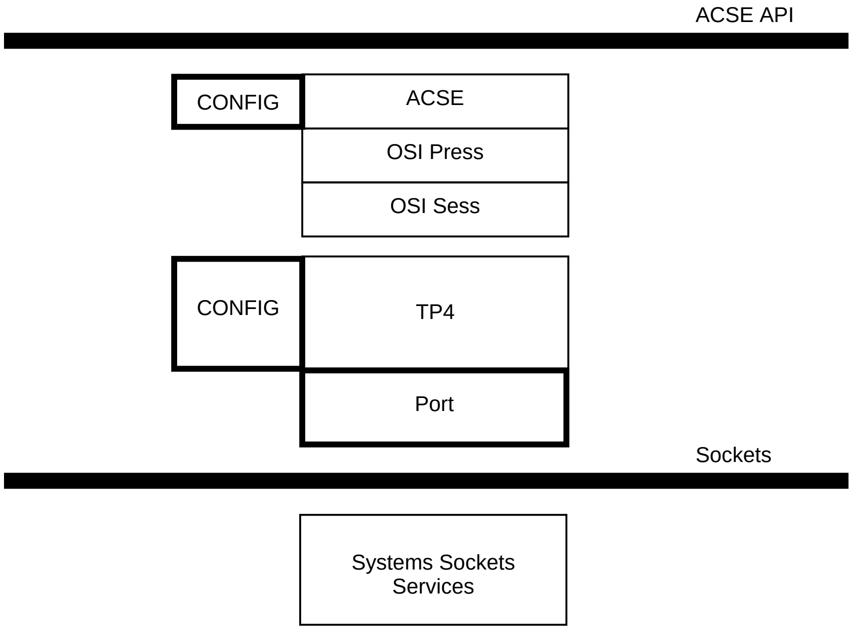
### All MVL Profiles



# 7 Layer OSI over Ethernet



# TCP/IP (via RFC1006)



# Lower Layer Component Portation

## OSI Transport Layer (TP4) Portation

For a new operating system or hardware platform, the following functions need to be ported. These are described in detail below.

`tp4_init_timer`

`tp4_check_timer`

### *Compile Time Options*

The following is a mandatory compile time option:

| | |
|---|---|
| `-DLEAN_T` | Compiles this version of TP4 API |

The following is an optional compile time option:

| | |
|---|---|
| `-DDEBUG_SISCO` | Enable logging using “slog” |

### tp4_init_timer

**Usage:** This function is called from the TP4 initialization (from **`tp4_initialize`**). This function should do anything necessary to initialize the timer service.

**Function Prototype:** `ST_VOID tp4_init_timer (ST_VOID);`

**Parameters:** NONE

**Return Value:** `ST_VOID`

## tp4_check_timer

**Usage:** This function is called from **`tp4_event`**. This function should check the timer and if one second has elapsed, it should call **`tp4_timer_tick`**. The example function, supplied by SISCO, is appropriate for most systems, but it may be freely modified if a more efficient approach is available on the target system.

On event driven systems, it is important to be sure that **`mvl_comm_serve`** is called at least once every second so that this function is also called. Therefore, the system should never go to sleep for more than one second. However, this restriction only applies if TP4 transport is used (i.e., the OSI/TP4 stack is used).

**Function Prototype:** `ST_VOID tp4_check_timer (ST_VOID);`

**Parameters:** NONE

**Return Value:** `ST_VOID`

**WARNING:** Do not call **`tp4_timer_tick`** from an interrupt handler. It must only be called from **`tp4_check_timer`**, which is only called by **`tp4_event`**.

# OSI Subnetwork Layer Portation

The user must provide the Subnetwork API. See *Appendix G* for more information.

# TCP/IP (via RFC1006)

The TCP/IP (via RFC1006) Protocol Stack is made up of the following components:

- ACSE
- OSI Presentation
- OSI Session
- TP0 (OSI Transport Class 0)
- TCP/IP (provided by the operating system with a Sockets interface)

## *Overview of Sockets Interface Implementation*

The sockets interface for MMS-EASE *Lite* consists of at least three tasks, each of which is blocking (waiting for a message from the user or from one of the other tasks). They are as follows:

| | |
|---|---|
| Main Task | (only one instance) |
| Listen Task | (Listens on socket. Only one instance) |
| Read Task | (Reads from socket. One instance for each connection) |

The Main Task includes all of the MMS encoding and decoding and the user interface. At startup, it spawns the Listen Task, which waits for incoming connections. When an incoming connection is detected, the Listen Task sends a pipe message to the Main Task, which calls “accept” to accept it, and then spawns an instance of the Read Task.

To make outgoing connections, the Main Task makes a non-blocking connect call (see **tp0_sock.c** and **`p_connect_req`**) and then spawns an instance of the Read Task, which waits for the connection to complete.

In either case (incoming or outgoing), when the connection phase is complete, the Read Task receives packets on the connection and passes them up to the Main Task. Sending packets on all connections is done from the Main Task.

The Main Task contains many layers of function calls, so it is sometimes difficult to trace how you get from the user level function down to the lowest layer. For example, the sequence of function calls for processing incoming events is as follows:

| | |
|---|---|
| `main` | (\mmslite\mvl\usr\server\server.c) calls |
| `mvl_comm_serve` | (\mmslite\mvl\src\mvl_serv.c) calls |
| `mvl_net_service` | (\mmslite\mvl\src\acse\mvl_acse.c) calls |
| `copp_event` | (\mmslite\uca\acse\acse2enc.c) calls |
| `tp0_event` | (\mmslite\uca\leant\tp0main.c) calls |
| `np_event` | (\mmslite\uca\leant\tp0_unix.c) |

If you look at **`np_event`**, you'll see that it first waits for a Pipe message from the Read Task (**`sreadd`**) or the Listen Task (**`slistend`**).

| | |
|---|---|
| `slistend` | (\mmslite\uca\leant\tp0_list.c) |
| `sreadd` | (\mmslite\uca\leant\tp0_read.c) |

**Note**: The tasks sreadd and slistend are spawned internally from the MMS-EASE Lite application. They are typically built in the **mmslite/bin** directory but are not part of the PATH environment variable by default. sreadd and slistend may either be put in the PATH environment variable by adding **/mmslite/bin** to the PATH or copied to a directory that is already in the PATH such as **/usr/bin**, **/usr/local/bin**, or **/bin** directories. If the MMS-EASE Lite application abnormally terminates the slistendd and sreadd must sometimes be manually terminated with a Unix kill command.

## *TCP/IP Porting*

The only porting that should be required is for the interface to TCP/IP provided by the operating system. This code has already been ported to some operating systems. Porting to other systems is simple if the system provides the following services:

- Sockets interface to TCP/IP
- Multitasking or Multithreading
- Pipes

***Note:*** *If your operating system does not support these services, contact SISCO Technical Support for further assistance.*

Only four modules should need to be modified (or replaced) for most ports. They are all located in **\mmslite\uca\leant** and are as follows:

| | |
|---|---|
| **tp0_sock.c** | (common sockets code. Part of Main Task.) |
| **tp0_list.c** | (Listen Task. UNIX & Win32 code.) |
| **tp0_read.c** | (Read Task. UNIX & Win32 code.) |
| **tp0_unix.c** | (UNIX-specific code. Part of Main Task) |
| OR | |
| **tp0_w32.c** | (Win32-specific code. Part of Main Task) |

## *Compile Time Options*

All of the code for the TCP/IP (via RFC1006) Stack is compiled into the **ositcpe.lib** library (**ositcpe.a** on UNIX-like systems). The following compile time options MUST be used so that the correct code is enabled in the following libraries:

```
-D LEAN_T
-D MOSI
-D TP0_ENABLED
```

The following is an optional compile time option:

| | |
|---|---|
| `-D DEBUG_SISCO` | Enable logging |

# Lower Layer Configuration

## OSI Transport Layer (TP4) Configuration

The user must fill in the following global structure to configure the TP4 API:

**TP_CFG tp_cfg;**

where TP_CFG is defined as follows:

```
typedef struct
  {
  ST_UINT16 max_tpdu_len;     /* max len of TPDU. Base on SNPDU size. */
                              /* Use to allocate TPDU buffers.        */
  ST_UCHAR max_tpdu_len_enc;  /* Binary encoded MAX TPDU len. Computed*/
                              /* from max_tpdu_len by tp4_initialize. */
  ST_UCHAR max_rem_cdt;       /* Max credits we can handle.           */
                              /* Will allocate this many TPDU_DT      */
                              /* structs.                             */
                              /* CRITICAL: MUST BE POWER OF 2.        */
  ST_UCHAR loc_cdt;           /* CDT value we will ALWAYS send in ACK */
                              /* We only accept in-sequence TPDUs so  */
                              /* only purpose of this is to           */
                              /* allow peer to send ahead.            */
  ST_UCHAR max_spdu_outst;
                              /* Max # of SPDUs outstanding per conn. */
                              /* Will allocate this many SPDU_INFO    */
                              /* structs for transmit queue.          */
                              /* CRITICAL: MUST BE POWER OF 2.        */
  ST_UCHAR max_num_conns;     /* Max # Connections                    */
  ST_UINT16 window_time;      /* Window Time                          */
  ST_UINT16 inact_time;       /* Inactivity Time                      */
  ST_UINT16 retrans_time;     /* Retransmission Time                  */
  ST_UCHAR max_trans;         /* Max # of transmissions of a TPDU     */
  ST_UCHAR ak_delay;          /* # of loops to delay sending AK.      */
  } TP_CFG;
```

The user must set each parameter before calling **tp4_initialize**. Behavior is undefined if this structure is modified after **tp4_initialize**. This may be done in any way appropriate for the target platform. An example of hard coding is provided in the following module:

**Tp4_hc.c**

An example of using the SISCO General Purpose Configuration Utility to configure the TP4 API, the CLNP API, as well as DIB entries is provided in the following module:

**lean_cfg.c**

This code processes the following configuration file:

**lean.cfg**

The configuration file (**lean.cfg**) is divided into three sections for TP4, CLNP, and DIB entries respectively. It is designed to divide into three separate files for applications that may not use all of the OSI protocol layers. A complete description of this file and the SISCO General Purpose Configuration Utility is beyond the scope of this document.

# TCP/IP Configuration

The user must fill in the following global structure to configure the TP0 API required for the **TCP/IP** (via RFC1006) stack:

```
TP0_CFG tp0_cfg;
```

where TP0_CFG is defined as follows:

```
typedef struct
  {
  ST_UINT16 max_tpdu_len;     /* max len of TPDU.                    */
                              /* Use to allocate TPDU buffers.       */
  ST_UCHAR max_tpdu_len_enc;  /* Binary encoded MAX TPDU len. Computed*/
                              /* from max_tpdu_len by tp0_initialize. */
  ST_UCHAR max_num_conns;     /* Max # Connections                   */
  ST_BOOLEAN keepalive;       /* Use KEEPALIVE option on Sockets.    */
  } TP0_CFG;                  /* For TP0/RFC1006 only.               */
```

The user must set each parameter before calling **tp0_initialize**. Behavior is undefined if this structure is modified after **tp0_initialize**. This may be done in any way appropriate for the target platform. An example of hard coding is provided in the following module:

**tp4_hc.c**

An example of using the SISCO General Purpose Configuration Utility to configure the TP4 API, the CLNP API, as well as DIB entries is provided in the following module:

**lean_cfg.c**

This code processes the following configuration file:

**lean.cfg**

The configuration file (**lean.cfg**) is divided into three sections for TP4, CLNP, and DIB entries respectively. It is designed to divide into three separate files for applications that may not use all of the OSI protocol layers. A complete description of this file and the SISCO General Purpose Configuration Utility is beyond the scope of this document.

# OSI Network Layer (CLNP/ES-IS) Configuration

The user must fill in the following global structure to configure the OSI Network (CLNP) API:

```
CLNP_PARAM clnp_param;
```

where CLNP_PARAM is defined as follows:

```
typedef struct
  {
  ST_UCHAR   pdu_lifetime;
                              /* PDU lifetime (in 500 msec units) for */
                              /* outgoing DT PDUs.                    */
                              /*   init to CLNP_DEF_PDU_LIFETIME      */
  ST_UCHAR   pdu_lifetime_dec;
                              /* PDU lifetime decrement (1=500msec)   */
                              /* for incoming DT or ER PDUs.          */
                              /*   init to CLNP_DEF_PDU_LIFETIME_DEC  */
  ST_UINT16 esh_cfg_timer;
                              /* How often we report our presence to  */
                              /* other network entities (in seconds)  */
                              /*   init to CLNP_DEF_ESH_CFG_TIMER     */
  ST_UINT16 esh_delay;        /* Delay time before first ESH is sent  */
                              /*   init to CLNP_DEF_ESH_DELAY         */
  ST_UCHAR   loc_mac  [CLNP_MAX_LEN_MAC];
                              /* Local MAC address                    */
                              /* For ADLC the NS-USER sets the loc_mac*/
                              /* DEBUG: Now the loc_mac has to match  */
                              /* the address in adlc.cfg !!!          */
                              /* For the Ethernet this param will be  */
                              /* read from the driver during init     */
  ST_UCHAR   loc_nsap [1+CLNP_MAX_LEN_NSAP];
                              /* Local len & NSAP address             */
  }CLNP_PARAM;
```

The user must set each parameter before calling **`clnp_init`**. Behavior is undefined if this structure is modified after **`clnp_init`**. This may be done in any way appropriate for the target platform. An example of hard coding is provided in the following module:

**clnp_hc.c**

An example of using the SISCO General Purpose Configuration Utility to configure the TP4 API, CLNP API, as well as DIB entries is provided in the following module:

**lean_cfg.c**

This code processes the following configuration file:

**lean.cfg**

The configuration file **lean.cfg** is divided into three sections for TP4, CLNP, and DIB entries respectively. It is designed to divide into three separate files for applications that may not use all of the OSI protocol layers. A complete description of this file and the SISCO General Purpose Configuration Utility is beyond the scope of this document.

# Network Addresses

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.

To configure the Network Addresses, the user must set the following global pointers to point to arrays of `DIB_ENTRY` structures:

| | |
|---|---|
| `DIB_ENTRY *loc_dib_table;` | Local Addresses (must have at least one) |
| `DIB_ENTRY *rem_dib_table;` | Remote Addresses (only needed for Client) |

```
typedef struct
  {
  ST_LONG reserved;          /* reserved field                      */
  ST_CHAR *name;             /* AR Name */
  ST_CHAR local;             /* SD_TRUE if local, SD_FALSE if remote*/
  ST_UCHAR AP_title_pres;    /* present flag                        */
  MMS_OBJ_ID AP_title;       /* AP title                            */
  ST_UCHAR AP_inv_id_pres;   /* present flag                        */
  ST_INT32 AP_invoke_id;     /* AP invocation ID                    */
  ST_UCHAR AE_qual_pres;     /* present flag                        */
  ST_INT32 AE_qual;          /* AE qualifier                        */
  ST_UCHAR AE_inv_id_pres;   /* present flag                        */
  ST_INT32 AE_invoke_id;     /* AE invocation ID                    */
  PRES_ADDR pres_addr;       /* Presentation address.               */
  } DIB_ENTRY;
```

This **DIB_ENTRY** definition references the **PRES_ADDR** structure defined below:

```
typedef struct tagPRES_ADDR
  {
  int psel_len;
  char psel [MAX_PSEL_LEN];
  int ssel_len;
  char ssel [MAX_SSEL_LEN];
  ST_INT tp_type;               /* Transport Type: TP_TYPE_TP4,      */
                                /* TP_TYPE_TCP, or TP_TYPE_TPX.      */
  int tsel_len;
  char tsel [MAX_TSEL_LEN];
  int nsap_len;
  char nsap [MAX_IP_ADDR_LEN];  /* If TP_TYPE_TP4, contains NSAP.    */
                                /* If TP_TYPE_TCP, contains IP addr. */
                                /* Only used for “remote” addresses. */

  } PRES_ADDR;
```

***Note:*** *Based on the review of current OSI agreements, the PSEL, SSEL and TSEL parameters are all being changed to a maximum of 4 bytes, improving the memory usage of MMS-EASE* Lite*. The standards recommend the following:*

*PSEL 4 - International Standard Profiles*
*SSEL 2 - GOSIP Ver2*
*TSEL 2 - GOSIP Ver2*

The transport type **TP_TYPE_TPX**, may be used only for a “local” entry. It indicates that both TP4 and TCP are to be supported.

Setting the pointers **loc_dib_table** and **rem_dib_table** may be done in any way appropriate for the target platform. Examples of hard-coding are provided in the following sample code modules (the code is executed only if HARD_CODED_CFG is defined):

**server.c**

**client.c**

An example of using the SISCO General Purpose Configuration Utility to configure the TP4 API, CLNP API, as well as DIB entries is provided in the following module:

**lean_cfg.c**

This code processes the following configuration file:

**lean.cfg**

The configuration file **lean.cfg** is divided into three sections for TP4, CLNP, and DIB entries respectively. It is designed to divide into 3 separate files for applications that may not use all of the OSI protocol layers. A complete description of this file and the SISCO General Purpose Configuration Utility is beyond the scope of this document.

# ACSE Authentication

The following describes the ACSE Authentication per Annex B of the ISO/IEC 8650-1.

The **acseauth.h** file contains the authentication structure **`ACSE_AUTH_INFO`** that is passed to/from the user and the ASN.1 parser.

If ACSE Authentication is not desired, the calling node may call **`mvla_initiate_req`** to send an initiate request PDU to the called node. If ACSE Authentication is needed, the ACSE user must call **`mvla_initiate_req_ex`** and pass a pointer to an **`ACSE_AUTH_INFO`** structure containing the authentication information they wish to send to the called node. The encoding of the authentication information is per the ACSE specification and is done in **acse2enc.c**.

The called side will receive the request PDU with authentication, and decode it in **acse2dec.c**. An **`ACSE_AUTH_INFO`** structure is filled out and passed to the user via **`u_mvl_connect_ind_ex`**. The user can accept the authentication, and return success, or reject for a variety of reasons, which will cause an abort PDU to be sent to the calling node. The reject reasons are a part of the constants in **acseauth.h** and are encoded in the abort PDU.

Also, the user is passed a pointer to a responding authentication structure, which may be sent back to the calling node during the connect confirm. Using this method of exchanging authentication information in both the associate request and associate response APDUs provides bi-directional authentication.

If the calling side does in fact receive authentication in the AARE APDU this information is passed to the user in **`u_mvl_connect_cnf_ex`**. Again, this function may return success or an error diagnostic, which will be encoded and sent in an abort PDU.

The authentication value itself is defined in the ACSE specification. The **`ACSE_AUTH_INFO`** structure may use a password mechanism (as defined in the ACSE spec) or some other mechanism. In the case of the “other” mechanism, the user is expected to handle the ASN.1 decoding and encoding of the authentication value. In addition, SISCO can provide certificate-based ACSE authentication mechanism.

ACSE authentication encoding/decoding is compiled into the MMS-EASE Lite library code. For ACSE Authentication sample code, please see the provided client, server, or uca_srvr in the **\mmslite\mvl\usr** directory.

Chapter 5

# MMS-EASE *Lite* Application Program Interfaces

MMS-EASE *Lite* has two distinct interfaces:

1. MMS-EASE Virtual Light (MVL) interface
2. MMS Protocol Encode/Decode interface

***Note:*** *Review the summary points for each interface before embarking on the application design process. SISCO recommends that the MVL interface be used for most applications.*

## MVL (MMS Client and Server Application Framework)

MMS-EASE *Lite* includes a higher-level interface layer referred to as MVL (**M**MS-**V**irtual-**L**ite). MVL is closely coupled to the lower layer subsystem components provided by SISCO and provides an application framework that is suitable for most applications.

MVL is integrated with all SISCO supplied network profiles, including the UCA profiles for Trim 7 and Reduced Stack, 7 Layer OSI, and TCP/IP (using RFC1006). MVL provides full integration with the SISCO ACSE layer, including the connection oriented and connectionless modes of operation. MVL allows the use of the Application Association object scope for connection oriented ACSE profiles.

For Server applications, application development is as simple as defining the MMS variables, variable lists, and types to be exposed to client applications then letting MVL do the rest. Hooks are provided to allow the application to participate in handling indications if desired, and MVL has the flexibility to handle most application programming requirements.

For Client applications, MVL provides an easy to use API for performing MMS connection control, Read, Write, and Identify services. Other services are easily added as required.

The most complete and accurate vehicle for developer documentation will be the MVL sample applications and the MVL header files. The Server sample is **\mmslite\mvl\usr\server\server.c** and the Client sample is **\mmslite\mvl\usr\client\client.c**. Most MVL features are demonstrated in these fully functional applications and most applications can easily be constructed using these samples as a starting point.

Advantages of the MVL interface include:

- MVL is a flexible application framework and provides useful general MMS services such as communications service, incoming PDU handling, etc.
- Works with SISCO's MMS Object Foundry, a utility that greatly simplifies creating and using MMS objects.
- Complete integration to the SISCO lower layer components (ACSE and below) is provided.
- Complete and flexible MMS object management code is provided, with an appropriate and conformant model.
- MMS Data conversion issues are addressed in a developer friendly manner.
- Fully functional Client and Server application examples are provided.
- Asynchronous response capability for the server.
- Multiple outstanding client request management is provided.
- This is the fastest way to get up and running.

# MMS Protocol Encode/Decode Subsystem

MMS-EASE *Lite* provides a MMS PDU Encode/Decode interface for those applications that are not a good fit with the MVL interface. For applications with non-standard network interface requirements and where there is a preference to address all the networking issues directly, this interface can be valuable. When using the PDU Encode/Decode subsystem, the user must address the following issues:

- How to manage MMS objects such as VMDs, Domains, Variables and Types.
- How to handle data conversion (MMS <-> Local formats).
- How to track outstanding requests (client only).
- What to do when a MMS Indication is received.
- How to work with the MMS-EASE data structures.
- How to interface to the lower layer communications subsystem; mapping of MMS onto the stack profile.
- How to handle multiple MMS connections.

Advantages of the MMS PDU Encode/Decode interface are:

- Simplicity
- Flexibility
- Small size
- No application framework to integrate into the target application.

## Chapter 6

# Using MVL

MVL (**M**MS **V**irtual **L**ite) is a communications framework for use with MMS-EASE *Lite* that is designed to speed implementation of complete MMS enabled systems. MVL currently supports a limited set of MMS services (see *MVL Services and Features* following this section), but can easily be extended to handle any number of services as client, server, or both.

MVL is easy to use and full featured samples of both client and server applications are included with the MMS-EASE Lite distribution. Reviewing these samples and the associated command and type definition files is the easiest way to begin working with the MMS-EASE *Lite* MVL interface.

MMS-EASE *Lite* is designed to operate as a single threaded application. After initialization, all MMS service is performed using the service function **`mvl_comm_serve`**. From within this call, all network service is performed including getting MMS PDUs from the network, decoding and operating on the MMS PDU, and calling any appropriate user functions. Note that global variables are used within MMS-EASE *Lite* and MVL, and so these functions are not reentrant.

MVL provides a MMS object framework such that development of a server application can be quite straightforward, requiring only application specific data types and variables to be integrated into the sample application. Generally, the application programmer can simply tell MMS-EASE *Lite* which variables are to be accessible via MMS and provide a data access mechanism.

# MVL Application Overview

## MVL Services and Features

MVL supports the most common services required for embedded client and server applications. The specific services currently supported are listed below. Note that it is a straightforward process to add additional services as required.

### *Services Supported*

| | |
|---|---|
| Initiate | Conclude |
| Cancel | Identify |
| Status | GetNameList |
| Read (Named, NamedVariableList) | Write (Named) |
| InformationReport (Named, NamedVariableList) | GetVariableAccessAttributes |
| GetVariableList Attributes | DefineVariableList |
| DeleteVariableList | FileDirectory |
| FileOpen | FileRead |
| FileClose | FileDelete |
| ReportJournalStatus | InitializeJournal |

# MVL Application Build Process

Many embedded environments require the use of a cross compiler, which runs on a *host* computer and the resulting programs are transferred to the *target* system for execution. In this discussion, the term *host* will refer to the environment where the application is compiled and linked, and *target* will refer to the environment where the application is to be executed.

The steps below are all to be executed on the host system and will result in an application that can be transferred to and executed on the target system. Note that this list assumes that the MMS-EASE *Lite* library build steps have already been successfully completed.

1. Create an MMS **O**bject **D**efinition **F**ile (ODF) for the application (named **srvrobj.odf** in the sample server applications). This text file is used to define all the MMS server objects and data types to be used by the application. It is then used by MMS Object Foundry to create C code that will be used to realize the objects. See *MMS Object Foundry* on page 265 for more information on MMS Object Foundry and Object Definition Files.
2. Create, compile, and link the sample application. Files to be included in the link include **mmsop_en.c**, **srvrobj.c**, **mvl_acse.c**, and MMS-EASE *Lite* libraries.

   See the MVL samples for make files for this process.

# Code Generation Utility Programs

MMS-EASE *Lite* includes two utility applications that are used to generate C source and header files to be used in the application. DOS and Win32 executable versions of these programs and associated source code are included with the MMS-EASE *Lite* distribution. Note that building these executables to run on the host will require building the MMS-EASE *Lite* libraries for the host environment as well as the target environment.

## *MMS Object Foundry*

This application is used to generate a C module for creating the MMS Objects for a MVL application. This executable takes as input an Object Definition File (ODF) which defines the MMS objects for the application, as well as a file describing the data alignment requirements for the target environment (**align.cfg**). See *MMS Object Foundry* on page 265 for more information regarding this utility program.

# Network Profiles

MMS-EASE *Lite* includes options for several stack profiles, including 7 Layer OSI, TCP/IP (via RFC1006), UCA Reduced Stack, and UCA Trim 7. These profiles all make use of SISCO’s ACSE as the upper interface, and so it is possible to develop the target application in such a manner as to be profile independent. When this is done, the developer simply selects the stack profile to be used by linking in alternate stack libraries. Of course, there will be some configuration differences between the various profiles.

MVL takes advantage of this common ACSE interface, using the MVL module **mvl_acse.c** as a bond between MVL and ACSE. Both connectionless and connection oriented operations are supported by MVL when available in the profile. MVL supports both CALLED and CALLING connection management with user hooks provided to allow the desired interaction with the application.

## Selecting MMS Services Set

The MMS-EASE *Lite* decode tree makes use of a set of function pointer and opcode control tables in the source module **mmsop_en.c**. The contents of these tables are controlled at compile time by the include file **mmsop_en.h**. This file must be edited to select the MMS PDUs to be decoded. This which must be done before the MMS source file **mmsop_en.c** is complied.

By default, MMS-EASE *Lite* will not support modifiers or companion standards, and will generate decode errors when they are encountered. To enable support for these elements, edit the **mmsintr2.c** file and define **`MOD_SUPPORT`** for Modifier support and **`CS_SUPPORT`** for Companion Standard support.

# MVL Configuration

MVL requires some configuration to perform as required by the application. Configuration for MMS-EASE *Lite* means initialization of memory based data structures. The items listed below are configurable. Note that additional configuration will be required for the selected stack profiles.

### *MMS Parameters*

**Maximum Message Size**

This is configured set by configuring the **`<Max_Mms_Pdu_Length>`** tag in the **osicfg.xml** file or by manually setting the **`max_msg_size`** parameter in the **`MVL_CFG_INFO`** structure, which is passed to the function **`mvl_start_acse`**.

This parameter represents the maximum MMS PDU size to be supported. This value is used for both calling and called connections and will impact the memory requirements of MVL. The MVL global variable **`mmsl_max_msg_size`** will be set to this value.

**Maximum Number Of Connections**

Setting the **`num_calling`** and **`num_called`** parameters in the **`MVL_CFG_INFO`** structure, which is passed to the function **`mvl_start_acse`**, controls the number of calling and called connections.

**MMS Services Supported**

The client and server service set and MMS parameter support items are configured by use of the header file **mmsop_en.h**. See *Appendix A* on page 303 for more information on using this file to control the service set.

**Other MMS Initiate Parameters**

The remaining MMS initiate parameters such as the number of outstanding requests, max structure nesting level, and MMS version are to be set dynamically by the user application when establishing a MMS connection.

### *Network Addressing*

**Local AR Names**

Before calling the **mvl_start_acse** startup function, the application needs to select the local AR Names to be used. These names are alias's for all required addressing for the node, and must be present in the applications DIB. See page 37 for more information on configuring AR Names.

For connection oriented ACSE, the local AR Name is set in the **MVL_CFG_INFO** structure, passed to the function **mvl_start_acse**. For connectionless ACSE, the local AR Name is passed to **mvl_init_audt_addr** to get the local address.

# MVL Connection Management

### *MVL Network Information Structure*

The following data structure is used to maintain information about a connection to a remote device. It represents the device the PDU is sent to or received from and is implementation specific.

```
typedef struct
  {
  struct mvl_aa_obj_ctrl *aa_objs; /* AA object ctrl                         */
  struct mvl_vmd_ctrl    *rem_vmd; /* Remote VMD                             */
  struct mvl_ind_pend    *pend_ind;
  ST_BOOLEAN conn_active;        /* Set SD_TRUE when the connection is up */
  ST_INT max_pdu_size;
  ST_INT index;                  /* NET_INFO table index for this elmnt  */

  ST_INT           maxpend_req;    /* num outstanding reqs negotiated    */
  ST_INT           numpend_req;    /* num reqs currently outstanding     */

#ifdef ICCP_LITE_SUPP
  ST_BOOLEAN mi_in_use;
  struct _mi_conn *mi_conn;
#endif
  INIT_INFO rem_init_info;       /* Services supported by remote device  */
  INIT_INFO locl_init_info;      /* Initiate info we sent                */
  AARQ_APDU ass_ind_info;
                                 /* Items below are used by MVL only     */
  ST_BOOLEAN in_use;             /* Flag that this 'NET_INFO' is in use  */
  ST_INT32 acse_conn_id;         /* ACSE's connection ID                 */

  ST_VOID *user_info;            /* MVL user can use this for 'whatever' */

  } MVL_NET_INFO;
```

**`struct mvl_aa_obj_ctrl *aa_objs`**

This pointer references the control structure containing all Application Association Specific objects associated with the connection.

**`struct mvl_vmd_ctrl    *rem_vmd`**

This pointer was used to receive InformationReports from the remote device. There is a different mechanism in place for receiving InformationReports and this pointer is no longer part of it. It is left in for backward compatibility. Please refer to function **u_mvl_info_rpt_ind** for further details on receiving Information Reports.

**`struct mvl_ind_pend    *pend_ind;`**

Used to reference outstanding pending indications.

**`ST_BOOLEAN conn_active`**

This field is set to **`SD_TRUE`** when the connection is up.

**`ST_INT max_pdu_size`**

This is the size of the largest MMS PDU, which may be sent or received from the remote device. It is negotiated between the two devices and may be less than the global variable **`mmsl_max_msg_size`**.

**`ST_INT index`**

This is the position of the **`MVL_NET_INFO`** structure in its global table.

**`ST_INT maxpend_req`**

The possible number of outstanding requests on this connection.

**`ST_INT numpend_req`**

The current number of outstanding requests on this connection.

**`ST_BOOLEAN mi_in_use`**

Not used by MMS-EASE *Lite*

**`struct _mi_conn *mi_conn`**

Not used by MMS-EASE *Lite*

**`INIT_INFO rem_init_info`**

This field contains the MMS Initiate information from the remote node. Among other things it includes the MMS service support string.

**`INIT_INFO locl_init_info`**

This field contains the local MMS Initiate information sent to the remote node.

**`AARQ_APDU ass_ind_info`**

This field contains the ACSE Application Request PDU information. Calling and called AP Title, AE Qualifier information may be found there.

**`ST_BOOLEAN in_use`**

This field tells when the **`MVL_NET_INFO`** structure is in use.

**`ST_INT32 acse_conn_id`**

This field contains the ACSE connection ID.

**`ST_VOID *user_info`**

This is reserved for application use and is not modified by MMS-EASE *Lite*.

## *MVL Functions*

### mvl_initiate_req

**Usage:** This synchronous function initiates a MMS connection to the selected Remote AR. The **remAr** name must be present in the **DIB_ENTRY** table.

**Function Prototype:**

```
ST_RET mvl_initiate_req (ST_CHAR *remAr,
                         INIT_INFO *req_info,
                         INIT_INFO *resp_info,
                         MVL_NET_INFO **net_info_out);
```

**Parameters:**

remAr — Remote AR Name (see Network Addresses on page 37.)

req_info — Proposed Initiate parameters (sent on request). The **INIT_INFO** structure is defined in **mms_pcon.h**.

resp_info — Negotiated Initiate parameters (received on response).

net_info_out — Pointer to pointer to connection control structure. The function allocates a **MVL_NET_INFO** structure and sets (***net_info_out**) to the address of the allocated structure. For example, if there is a variable **MVL_NET_INFO *net_info**, and **&net_info** is passed to the function, it will set **net_info** to the address of the new structure. The **MVL_NET_INFO** structure is defined in **mvl_defs.h**.

**Return Value:** ST_RET SD_SUCCESS If OK, or an error code.

## mvla_initiate_req

**Usage:** This synchronous function is similar to the synchronous versions, except that the return before the confirm has been received and instead returns a **`MVL_REQ_PEND`** request control structure. When the confirm is received, the **`u_req_done`** function pointer element in the request control structure is invoked, at which time the user can examine the response information.

**Function Prototype:**

```
ST_RET mvla_initiate_req (ST_CHAR *remAr,
                          INIT_INFO *req_info,
                          INIT_INFO *resp_info,
                          MVL_NET_INFO **net_info_out,
                          MVL_REQ_PEND **req_out);
```

**Parameters:**

`remAr` Remote AR Name (see “Network Address Configuration” section)

`req_info` Proposed Initiate parameters (sent on request). The **`INIT_INFO`** structure is defined in **mms_pcon.h**.

`resp_info` Negotiated Initiate parameters (received on response).

`net_info_out` Pointer to pointer to connection control structure. The function allocates a **`MVL_NET_INFO`** structure and sets **`(*net_info_out)`** to the address of the allocated structure. For example, if there is a variable **`MVL_NET_INFO *net_info`**, and **`&net_info`** is passed to the function, it will set **`net_info`** to the address of the new structure. The **`MVL_NET_INFO`** structure is defined in **mvl_defs.h**.

`req_out` Pointer to pointer to request control structure. The function allocates a **`MVL_REQ_PEND`** structure and sets **`(*req_out)`** to the address of the allocated structure. For example, if there is a variable **`MVL_REQ_PEND *req_pend`**, and **`&req_pend`** is passed to the function, it will set **`req_pend`** to the address of the new structure. The **`MVL_NET_INFO`** structure is defined in **mvl_defs.h**. The structure must be freed sometime after the response is received and processed by calling **`mvl_free_req_ctrl`**(**`req_pend`**).

**Return Value:** `ST_RET` `SD_SUCCESS` If request sent successfully, or an error code.

## mvla_initiate_req_ex

**Usage:** This function initiates a MMS connection to the selected Remote AR. The **`remAr`** name must be present in the **`DIB_ENTRY`** table.

**Function Prototype:**

```
ST_INT mvla_initiate_req_ex (ST_CHAR *remAr,
                             INIT_INFO *req_info,
                             INIT_INFO *resp_info,
                             MVL_NET_INFO **net_info_out,
                             MVL_REQ_PEND **req_out,
                             ACSE_AUTH_INFO *auth_info,
                             S_SEC_ENCRYPT_CTRL *encrypt_info);
```

**Parameters:**

`remAr` Remote AR Name (see Network Addresses on page 37.)

`req_info` Pointer to the proposed Initiate parameters (sent on request). The **`INIT_INFO`** structure is defined in **mms_pcon.h**.

`resp_info` Pointer to the negotiated Initiate parameters (received on response).

`net_info_out` Pointer to pointer to connection control structure. The function allocates a **`MVL_NET_INFO`** structure and sets (**`*net_info_out`**) to the address of the allocated structure. For example, if there is a variable **`MVL_NET_INFO *net_info`**, and **`&net_info`** is passed to the function, it will set **`net_info`** to the address of the new structure. The **`MVL_NET_INFO`** structure is defined in **mvl_defs.h**.

`req_out` Pointer to pointer to request control structure. The function allocates a **`MVL_REQ_PEND`** structure and sets **`(*req_out)`** to the address of the allocated structure. For example, if there is a variable **`MVL_REQ_PEND *req_pend`**, and **`&req_pend`** is passed to the function, it will set **`req_pend`** to the address of the new structure. The **`MVL_NET_INFO`** structure is defined in **mvl_defs.h**. The structure must be freed sometime after the response is received and processed by calling **`mvl_free_req_ctrl`**(**`req_pend`**).

`auth_info` Pointer to structure containing ACSE Authentication information for this connection.

`encrypt_info` For future implementation – currently must be NULL.

**Return Value:** `ST_RET` `SD_SUCCESS` If OK, or an error code.

## mvl_concl

**Usage:** This is a synchronous function for sending a MMS Conclude. It will not return until the response has been received or it gives up.

**Function Prototype:**

```
ST_RET mvl_concl (MVL_NET_INFO *net_info,
                  MVL_REQ_PEND **req_out);
```

**Parameters:**

| | |
|---|---|
| net_info | Network connection information. |
| req_out | See the description of **req_out** on page 183. |

**Return Value:** ST_RET SD_SUCCESS If OK, or an error code.

## mvla_concl

**Usage:** This is an asynchronous function for sending a MMS Conclude.

**Function Prototype:**

```
ST_RET mvla_concl (MVL_NET_INFO *net_info,
                   MVL_REQ_PEND **req_out);
```

**Parameters:**

| | |
|---|---|
| net_info | Network connection information. |
| req_out | See the description of **req_out** on page 183. |

**Return Value:** ST_RET SD_SUCCESS If OK, or an error code.

## u_mvl_concl_ind

**Usage:** This is a user function called by MVL when a conclude indication is received. It should do all appropriate cleanup before sending the Conclude response. At a minimum, it should call **mplas_concl_resp** to send the response. See the file **server.c** for an example of this function.

**Function Prototype:**

```
ST_VOID u_mvl_concl_ind (MVL_COMM_EVENT *event);
```

**Parameters:**

| | |
|---|---|
| event | This is the communications event control structure. |

**Return Value:** ST_VOID

## mplas_concl_resp

**Usage:** This function sends the Conclude response.

**Function Prototype:** `ST_VOID mplas_concl_resp (MVL_COMM_EVENT *event);`

**Parameters:**

| | |
|---|---|
| `event` | This is the communications event control structure. |

**Return Value:** `ST_VOID`

## mvl_abort_req

**Usage:** This function is used to abort a MMS connection. It causes abrupt termination of the connection.

**Function Prototype:** `ST_RET mvl_abort_req (MVL_NET_INFO *net_info);`

**Parameters:**

| | |
|---|---|
| `net_info` | Network connection information. |

| | | | |
|---|---|---|---|
| **Return Value:** | `ST_RET` | `SD_SUCCESS` | If OK, or an error code. |

## mvl_abort_req_ex

**Usage:** This function is used to abort a MMS connection. It causes abrupt termination of the connection.

**Function Prototype:**

```
ST_RET mvl_abort_req_ex (MVL_NET_INFO *cc,
                         ST_BOOLEAN diagnostic_pres,
                         ST_ACSE_AUTH diagnostic);
```

**Parameters:**

`cc` Pointer to the network connection information.

`diagnostic_pres` Flag indicating whether the **`diagnostic`** value should be sent in an Abort PDU.

`diagnostic` Diagnostic value to sent in an Abort PDU. Must be one of the following:

| | |
|---|---|
| ACSE_AUTH_SUCCESS | 0 |
| ACSE_DIAG_NO_REASON | 1 |
| ACSE_DIAG_PROTOCOL_ERR0R | 2 |
| ACSE_DIAG_AUTH_MECH_NAME_NOT_RECOGNIZED | 3 |
| ACSE_DIAG_MECH_NAME_REQUIRED | 4 |
| ACSE_DIAG_AUTH_FAILURE | 5 |
| ACSE_DIAG_AUTH_REQUIRED | 6 |

**Return Value:** `ST_RET` `SD_SUCCESS` or error code

## mvl_release_req

**Usage:** This function is used to release a MMS connection. It causes an orderly termination of the connection and should always be preceded by a successful MMS conclude sequence.

**Function Prototype:**

```
ST_RET mvl_release_req (MVL_NET_INFO *net_info);
```

**Parameters:**

`net_info` Network connection information.

**Return Value:** `ST_RET` `SD_SUCCESS` If OK, or an error code.

## mvla_release_req

**Usage:** This function is similar to the synchronous versions, except that that return before the confirm has been received and instead returns a **`MVL_REQ_PEND`** request control structure. When the confirm is received, the **`u_req_done`** function pointer element in the request control structure is invoked, at which time the user can examine the response information.

**Function Prototype:**

```
ST_RET mvla_release_req (MVL_NET_INFO *net_info,
                         MVL_REQ_PEND **req_out);
```

**Parameters:**

`net_info` Network connection information.

`req_out` The user must pass the address of a variable of type (**`MVL_REQ_PEND *`**) to the function. The function allocates a **`MVL_REQ_PEND`** structure and sets the user's variable to the address of the allocated structure. For example, if the user has a variable **`MVL_REQ_PEND *req_pend`**, they should pass **`&req_pend`** to the function and it will set the value of **`req_pend`**. The user must free the structure sometime after the response is received and processed by calling **`mvl_free_req_ctrl`**(**`req_pend`**).

**Return Value:** `ST_RET` `SD_SUCCESS` If request sent successfully, or an error code.

## u_mvl_connect_ind_ex

**Usage:** This is a user-defined function that must handle connect indications.

| | |
|---|---|
| ***IMPORTANT NOTICE:*** | Unlike the previously used function pointer, **u_mvl_connect_ind_fun**, this function is required to be implemented by the user. It may break some older existing applications that will refuse to link until this function is implemented. |

**Function Prototype:**

```
ST_ACSE_AUTH u_mvl_connect_ind_ex (MVL_NET_INFO *cc,
                                   INIT_INFO *init_info,
                                   ACSE_AUTH_INFO *req_auth_info,
                                   ACSE_AUTH_INFO *rsp_auth_info);
```

**Parameters:**

`cc` Pointer to the network connection information. The **MVL_NET_INFO** structure is defined in **mvl_defs.h**.

`init_info` Proposed Initiate parameters. The **INIT_INFO** structure is defined in **mms_pcon.h**.

`req_auth_info` A pointer to the **ACSE_AUTH_INFO** structure that was received from the calling partner. Please see the MVL sample server (**server.c**) in the function **u_mvl_connect_ind_ex** for a sample of how to use password-based ACSE authentication.

`rsp_auth_info` A pointer to the **ACSE_AUTH_INFO** structure that will be encoded and returned to the calling partner. Please see the MVL sample server (**server.c**) in the function **u_mvl_connect_ind_ex** for a sample of how to use password-based ACSE authentication.

**Return Value:**

`ST_ACSE_AUTH` One of the following defined values. If the return value is **ACSE_AUTH_SUCCESS**, a positive Initiate response is sent to the calling node. If the return value is not **ACSE_AUTH_SUCCESS**, an ACSE Abort PDU is sent using the return value as the ABRT-diagnostic.

```
#define ACSE_AUTH_SUCCESS                       0
#define ACSE_DIAG_NO_REASON                     1
#define ACSE_DIAG_PROTOCOL_ERROR                2
#define ACSE_DIAG_AUTH_MECH_NAME_NOT_RECOGNIZED 3
#define ACSE_DIAG_AUTH_MECH_NAME_REQUIRED       4
#define ACSE_DIAG_AUTH_FAILURE                  5
#define ACSE_DIAG_AUTH_REQUIRED                 6
```

## u_mvl_connect_cnf_ex

**Usage:** This is a user-defined function that must handle connect confirms.

**Function Prototype:**

```
ST_ACSE_AUTH u_mvl_connect_cnf_ex (MVL_NET_INFO *cc,
                                   AARE_APDU *assoc_rsp_info);
```

**Parameters:**

| | |
|---|---|
| `cc` | Pointer to the network connection information. The **`MVL_NET_INFO`** structure is defined in **mvl_defs.h**. |
| `assoc_rsp_info` | A pointer to the **`ACSE_AUTH_INFO`** structure that was received from the called partner. Please see the MVL sample client (**client.c**) in the function **`u_mvl_connect_cnf_ex`** for a sample of how to use password-based ACSE authentication. |

**Return Value:**

`ST_ACSE_AUTH` One of the following defined values. If the return value is not **`ACSE_AUTH_SUCCESS`**, an Abort PDU is sent using  the return value as the ABRT-diagnostic.

```
#define ACSE_AUTH_SUCCESS                      0
#define ACSE_DIAG_NO_REASON                    1
#define ACSE_DIAG_PROTOCOL_ERROR               2
#define ACSE_DIAG_AUTH_MECH_NAME_NOT_RECOGNIZED 3
#define ACSE_DIAG_AUTH_MECH_NAME_REQUIRED      4
#define ACSE_DIAG_AUTH_FAILURE                 5
#define ACSE_DIAG_AUTH_REQUIRED                6
```

### u_mvl_disc_ind_fun

**Usage:** This is a user defined function pointer that handles disconnect indications.

```
#define MVL_ACSE_RELEASE_IND     1
#define MVL_ACSE_ABORT_IND       2
```

**Function Pointer Global Variable:**

```
extern ST_VOID (*u_mvl_disc_ind_fun)
                MVL_NET_INFO *net_info,
                ST_INT discType);
```

**Parameters:**

| | |
|---|---|
| `net_info` | This is the Network connection information. |
| `discType` | Indicates the type of disconnect. **`MVL_ACSE_RELEASE_IND`** if release, **`MVL_ACSE_ABORT_IND`** if abort. |

**Return Value:** `ST_VOID`

# Using MVL with MMS *Lite* ACSE Components

MVL is fully integrated with the MMS-EASE *Lite* ACSE components. This integration provides the MVL application developer with an easy to use mechanism for managing MMS connections. The MVL sample client application demonstrates the use of calling connections and the MVL sample server application demonstrates the use of called connections.

## Connection Management

MVL provides full ACSE connection management facilities via the source module **mvl_acse.c**. The MVL function **`mvl_start_acse`** must be called in order to initialize the lower layer subsystem. Prior to exiting the application, the MVL function **`mvl_end_acse`** should be called.

## Building mvl_acse

The MVL source module **mvl_acse.c** must be compiled and linked to the application in order to access the ACSE functionality of MMS-EASE *Lite*. Compile time switches are used to control the connection management capabilities of this module. The **`MMS_INIT_EN`** define in the file **mmsop_en.h** is used to control the inclusion of calling and/or called code.

The module **mvl_acse.c** will provide a table of **`MVL_NET_INFO`** data structures for calling and called connection management (e.g., **`mvl_calling_conn_ctrl`** and **`mvl_called_conn_ctrl`**), that can be referenced by the user application in managing connections as required.

# Being a Called Node

When an ACSE associate indication is received, MVL will parse the user information field looking for a MMS Initiate PDU. If one is present and can be decoded correctly, the user function **`u_mvl_connect_ind_ex`** is called. If the user returns **`ACSE_AUTH_SUCCESS`** from this function, MVL will respond positively to the Initiate using the Initiate response information provided by the global pointer **`mvl_init_resp_info`**. If the user returns any other value, an ACSE Abort PDU is sent using this value as the ABRT-Diagnostic.

# Connection Activity Notifications

The user may set the following function pointer in order to be notified when a Conclude or Abort indication is received:

```
u_mvl_disc_ind_fun
```

See the sample server application for an example of how this function pointer may be used.

# Extending the MVL Service Set

## MVL Server: Adding Support for another Service

To add another server service to the existing code framework, the following steps should be followed. This is most easily accomplished by selecting a similar existing service and using it as a template.

1. Modify the function `mvl_ind_rcvd` in **mvlop_en.c** to check for the new opcode and call a new processing function `mvl_process_xxx_ind`. The opcode defines can be found in **mms_def2.h**, located in the **\mmslite\inc** directory.
2. Edit **mvl_defs.h** and add the function prototype for the new `mvl_process_xxx_ind` function that will be used to process the indication.
3. Create a new module to contain the indication processing function. Copy an existing **s_xxxx.c** module, such as **s_ident.c**, and modify the `mvl_process_xxx_ind` function name and code as appropriate. Note that the MMS service aspect must be handled by the new indication processing function. That is, the requested MMS service activity must be correctly carried out per the MMS services specification.
4. In the server application, modify **mmsop_en.h** to enable decode of the indication for the new service (the define should be either `RESP_EN or REQ_RESP_EN`, depending if the application will also act as a client for the service).
5. Make any required changes to the MVL library make files and to the server application to support the new service.

## MVL Client: Adding Support for Another Service

To add another client service to the existing code framework, the following steps should be followed:

1. Modify **mmsop_en.h** to enable decode of the confirm for the new service. The define should be either `REQ_EN` or `REQ_RESP_EN`, depending if the application will also act as a server for the service.
2. Modify the function `mvl_conf_rcvd` in **mvlop_en.c** to check for the new opcode and call a new processing function `mvl_process_xxx_conf`. The opcode defines can be found in **mms_def2.h**, in the **\mmslite\inc** directory.
3. Create a new module to contain the confirm processing function. Copy an existing **c_xxxx.c** module (**c_ident.c** or **c_read.c**) and modify the `mvl_process_xxx_conf` function name and code as appropriate.
4. Add the `mvl_process_xxxx_conf` function prototype to **mvl_defs.h**.

# MVL Support Functions

## *Communication Service Functions*

### mvl_comm_serve

**Usage:** MVL Communication Service is a function that should be called periodically by the application. It will check for communications events and act on them, which will include decoding MMS PDUs and calling service functions. The mechanism used to determine when this function should be called is system specific and will depend on the lower layer service provider. This should be done at least once per second and whenever a low level network event is detected. The detection and use of network events is to be addressed during the porting phase.

**Function Prototype:** `ST_BOOLEAN mvl_comm_serve (ST_VOID);`

**Parameters:** NONE

| | | | |
|---|---|---|---|
| **Return Value:** | ST_BOOLEAN | SD_TRUE | If there is more communication service to be done (i.e., **`mvl_comm_serve`** should be called again). |
| | | SD_FALSE | **`mvl_comm_serve`** does not need to be called until one-second elapses or a communication event is detected. |

**NOTES:**

1. **Server Considerations**

   Once the MVL object configuration is complete, most services are handled transparently for the user, and any user code does not need to be directly involved. One area where the user application may be involved is in variable access, via the MVL pre/post processing functions for variables.

2. **Client Considerations**

   If the user application makes use of asynchronous client request functions (such as **`mvla_read_variables`**), the **`u_req_done`** callback function from the **`MVL_REQ_PEND`** structure will be invoked (if not **`NULL`**) from within the **`mvl_comm_serve`** function.

3. **ACSE Considerations**

   If the user application sets the ACSE disconnect callback function pointer (**`u_mvl_disc_ind_fun`**), the function will be invoked from within the **`mvl_comm_serve`** function.

## *Type Management Functions*

### mvl_init_type_ctrl

**Usage:** This function is used to initialize the MVL type control subsystem. It must be called before any communications activity can take place. This function is in the source module produced by the MMS Object Foundry.

**Function Prototype:**
```
ST_VOID mvl_init_type_ctrl (ST_VOID);
```

**Parameters:** NONE

**Return Value:** `ST_VOID`

### mvl_get_runtime

**Usage:** This function takes the type ID and provides a pointer to the runtime type and its size as output.

**Function Prototype:**
```
ST_RET mvl_get_runtime (ST_INT type_id,
                        RUNTIME_TYPE **rt_ptr_out,
                        ST_INT *num_rt_out);
```

**Parameters:**

| | |
|---|---|
| `type_id` | This is the MMS-EASE *Lite* type ID for which the Runtime Type is to be returned. |
| `rt_ptr_out` | This output parameter references the beginning of the runtime type array. |
| `num_rt_out` | This output parameter indicates the number of runtime type elements in the runtime type array. |

**Return Value:** `ST_RET` `SD_SUCCESS`

## mvl_mod_arr_size

**Usage:** This function can be used to modify the number of elements in an array of runtime types. For instance, this can be useful to avoid having to define all possible array Runtime types for alternate access support. The size of the array may be increased or decreased.

**Function Prototype:**
```
ST_VOID mvl_mod_arr_size (RUNTIME_TYPE *rt,
                          ST_INT num_elmnts);
```

**Parameters:**

| | |
|---|---|
| `rt` | This is a pointer to the Runtime type to be modified. |
| `num_elmnts` | This is the new value for the number of elements in the array. |

**Return Value:** `ST_VOID`

## *ACSE Interface Functions*

### MVL_CFG_INFO

The ACSE interface functions make use of the following structure:

```
typedef struct
{
ST_INT num_calling;                  /* number of calling connections */
ST_INT num_called;                   /* number of called connections  */
ST_INT max_msg_size;                 /* Max MMS message size          */
ST_CHAR local_ar_name[MAX_AR_LEN+1]; /* Local AR Name                 */
} MVL_CFG_INFO;
```

## osicfgx

**Usage:** This function reads the XML file that contains configuration parameters for MVL and the OSI Stack.

**Function Prototype:**
```
ST_RET osicfgx (ST_CHAR *xml_filename,
                MVL_CFG_INFO *mvlCfg);
```

**Parameters:**

| | |
|---|---|
| `xml_filename` | This is the name of the XML file to read. |
| `mvlCfg` | This is a pointer to a user structure containing parameters that are filled in by this function. |

**Return Value:** `ST_RET` `SD_SUCCESS` or error code.

## mvl_start_acse

**Usage:** This function is used to start the MVL ACSE subsystem.

**Function Prototype:** `ST_RET mvl_start_acse (MVL_CFG_INFO *mvlCfg);`

**Parameters:**

| | |
|---|---|
| `mvlCfg` | This is a pointer to a user structure containing parameters that are used to configure MVL. The structure must be filled in by calling **`osicfgx`** or by some other means. |

| | | |
|---|---|---|
| **Return Value:** | `ST_RET` | `SD_SUCCESS` or error code. |

## mvl_end_acse

**Usage:** This function is used to terminate the MVL ACSE subsystem.

**Function Prototype:** `ST_RET mvl_end_acse (ST_VOID);`

**Parameters:** NONE

**Return Value:** `ST_VOID`

### *Miscellaneous Functions*

## mvl_find_dom

**Usage:** This function is used to find a MVL Domain.

**Function Prototype:** `MVL_DOM_CTRL *mvl_find_dom (ST_CHAR *name);`

**Parameters:**

| | |
|---|---|
| `name` | Name of domain to find. |

| | | |
|---|---|---|
| **Return Value:** | `MVL_DOM_CTRL *` | Pointer to the Domain object. **`NULL`** if not found. The structure **`MVL_DOM_CTRL`** is defined in **mvl_defs.h**. |

## mvl_find_jou

**Usage:** This function is used to find a MVL Journal object given the MMS Object Name, which includes scope information.

**Function Prototype:** `MVL_JOURNAL_CTRL *mvl_find_jou (OBJECT_NAME *obj_name);`

**Parameters:**

| | |
|---|---|
| `obj_name` | The MMS Object Name of the Journal object to find. The structure **`OBJECT_NAME`** is defined in **mms_mp.h**. |

| | | |
|---|---|---|
| **Return Value:** | `MVL_JOURNAL_CTRL *` | Pointer to the Journal object. **`NULL`** if not found. The structure **`MVL_JOURNAL_CTRL`** is defined in **mvl_defs.h**. |

## mvl_find_nvl

**Usage:** This function is used to find a MVL Named Variable List object given the MMS Object Name, which includes scope information.

**Function Prototype:** `MVL_NVLIST_CTRL *mvl_find_nvl (OBJECT_NAME *obj_name);`

**Parameters:**

| | |
|---|---|
| `obj_name` | The MMS Object Name of the Named Variable List object to find. The structure **`OBJECT_NAME`** is defined in **mms_mp.h**. |

| | | |
|---|---|---|
| **Return Value:** | `MVL_NVLIST_CTRL *` | Pointer to the Named Variable List object. **`NULL`** if not found. The structure **`MVL_NVLIST_CTRL`** is defined in **mvl_defs.h**. |

## mvl_find_va

**Usage:** This function can be used to find a MVL Variable Association given the MMS Object Name, which includes scope information.

**Function Prototype:** `MVL_VAR_ASSOC *mvl_find_va (OBJECT_NAME *obj_name);`

**Parameters:**

`obj_name` The MMS Object Name of the Named Variable object to find. The structure **`OBJECT_NAME`** is defined in **mms_mp.h**.

**Return Value:** `MVL_VAR_ASSOC *` Pointer to the Named Variable object. **`NULL`** if not found. The structure **`MVL_VAR_ASSOC`** is defined in **mvl_defs.h**.

### *Manufactured Object Processing Functions*

## u_mvl_get_va_aa

**Usage:** The function will be called when a variable is being read or written and it is not present in the MVL Variable Association control tables. The user application can resolve the association and return a **`MVL_VARIABLE_ASSOCIATION`** if appropriate. Note that this function is only used when MVL is compiled with **`MVL_AA_SUPP`** and **`USE_MANUFACTURED_OBJ`** defined.

If **`*alt_access_done_out`** is set **`SD_TRUE`**, MVL will assume that the alternate access operation has been addressed by the called function.

**Function Prototype:**

```
MVL_VAR_ASSOC *u_mvl_get_va_aa (ST_INT service,
                                OBJECT_NAME *obj,
                                MVL_NET_INFO *netInfo,
                                ST_BOOLEAN alt_access_pres,
                                ALT_ACCESS *alt_acc,
                                ST_BOOLEAN *alt_access_done_out);
```

**Parameters:**

`service` The MMS service requiring VariableAccess look up. Values may be **`MMSOP_WRITE`**, **`MMSOP_MVLU_RPT_VA`**, **`MMSOP_INFO_RPT`**, or **`MMSOP_GET_VAR`**.

`obj` The name and scope of the variable needing to be resolved by the application.

`netInfo` A pointer to connection information, this provides the application with the means to resolve ApplicationAssociation specific variables. The structure **`MVL_NET_INFO`** is defined in **mvl_defs.h.**

## u_mvl_get_va_aa (cont'd)

**Parameters (cont'd):**

| | |
|---|---|
| alt_access_pres | Tells the application if AlternateAccess information is present. Values are **SD_TRUE** and **SD_FALSE**. |
| alt_acc | When the **alt_access_pres** parameter is set to **SD_TRUE**, this points to AlternateAccess information for the application to use when performing the VariableAccess. |
| alt_access_done_out | When the **alt_access_pres** parameter is set to **SD_TRUE**, this is set by the application if AlternateAccess is performed by the application. |

| | | |
|---|---|---|
| **Return Value:** | !=NULL | The application successfully manufactured the variable association. |
| | NULL | An error meaning the application did not resolve the variable. |

## u_mvl_free_va

**Usage:** When MVL is done using a manufactured Variable Association, it will call a user function selected by this function to allow the user to free the associated resources. Note that this function is only used when MVL is compiled with **USE_MANUFACTURED_OBJ** defined.

**Function Prototype:**

```
ST_VOID u_mvl_free_va (ST_INT service,
                       MVL_VAR_ASSOC *va,
                       MVL_NET_INFO *netInfo);
```

**Parameters:**

| | |
|---|---|
| service | The MMS service passed to the application when the **MVL_VAR_ASSOC** was manufactured. Values may be **MMSOP_WRITE**, **MMSOP_MVLU_RPT_VA**, **MMSOP_INFO_RPT**, or **MMSOP_GET_VAR**. |
| va | A pointer to the data structure originally returned from the application. |
| netInfo | A pointer to connection information associated with the variable that provides the application with the means to resolve ApplicationAssociation specific variables. The structure **MVL_NET_INFO** is defined in **mvl_defs.h** |

**Return Value:** ST_VOID

## u_mvl_get_nvl

**Usage:** This function will be called when a NamedVariableList is being read or written and it is not present in the MVL NamedVariableList control tables. The user application can resolve the association and return a **`MVL_NVLIST_CTRL`** if appropriate. **`NULL`** is returned when the NamedVariableList is unrecognized. Note that this function is only used when MVL is compiled with **`USE_MANUFACTURED_OBJ`** defined.

**Function Prototype:**

```
MVL_NVLIST_CTRL *u_mvl_get_nvl (ST_INT service,
                                OBJECT_NAME *obj,
                                MVL_NET_INFO *netInfo);
```

**Parameters:**

`service` The service that is referencing the NamedVariableList object. Possible values are **`MMSOP_GET_VLIST`** and **`MMSOP_READ`**.

`obj` The MMS Object Name of the NamedVariableList object to find. The structure **`OBJECT_NAME`** is defined in **mms_mp.h**.

`netInfo` A pointer to connection information associated with the NamedVariableList, this provides the application with the means to resolve ApplicationAssociation specific variables. The structure **`MVL_NET_INFO`** is defined in **mvl_defs.h**

**Return Value:**

| | |
|---|---|
| `!=NULL` | The application successfully manufactured the NamedVariableList. |
| `NULL` | An error meaning the application did not manufacture the NamedVariableList. |

## u_mvl_free_nvl

**Usage:** When MVL is done using a manufactured NamedVariableList control, it will call a user function selected by this function to allow the user to free the associated resources. Note that this function is only used when MVL is compiled with **`USE_MANUFACTURED_OBJ`** defined.

**Function Prototype:**
```
ST_VOID u_mvl_free_nvl (ST_INT service,
                        MVL_NVLIST_CTRL *nvl,
                        MVL_NET_INFO *netInfo);
```

**Parameters:**

`service` The MMS service passed to the application when the **`MVL_NVLIST_CTRL`** was manufactured. Possible values are **`MMSOP_GET_VLIST`** and **`MMSOP_READ`**.

`nvl` A pointer to the data structure originally returned from the application.

`netInfo` A pointer to connection information associated with the NamedVariableList that provides the application with the means to resolve ApplicationAssociation specific variables. The structure **`MVL_NET_INFO`** is defined in **mvl_defs.h**

**Return Value:** `ST_VOID`

## u_gnl_ind_vars

**Usage:** When the application is making use of the manufactured object approach, it will also be necessary to provide the list of objects to be returned when a MMS GetNameList indication is received. The user must provide this function. Note that this function is only called when MVL is compiled with **`USE_MANUFACTURED_OBJ`** defined.

**Function Prototype:**

```
ST_INT u_gnl_ind_vars (NAMELIST_REQ_INFO *req_info,
                       ST_CHAR **ptr,
                       ST_BOOLEAN *moreFollowsOut,
                       ST_INT maxNames);
```

**Parameters:**

`req_info` GetNameList request information. It is necessary to examine this structure to determine if there is a name to continue after. The structure **`NAMELIST_REQ_INFO`** is defined in **mms_pvmd.h**.

`ptr` An array of pointers to variable name character strings. The user must fill in the array.

`moreFollowsOut` Set **`*moreFollowsOut`** = **`SD_TRUE`** if not all variable names being manufactured by the application will fit in the NameList response. The maximum number of variable names that may be returned is supplied as the parameter **`maxNames`**. Set **`*moreFollowsOut`** = **`SD_FALSE`** when the function reports the last known manufactured variable name in the set.

`maxNames` The maximum number of variable names that may be returned by the function per call.

**Return Value:** `ST_INT` The number of variable names returned in the pointer table. 0 indicates that the function did not return any manufactured variable names.

## u_gnl_ind_nvls

**Usage:** When the application is making use of the manufactured object approach, it will also be necessary to provide the list of objects to be returned when a MMS GetNameList indication is received. The user must provide this function. Note that this function is only called when MVL is compiled with **USE_MANUFACTURED_OBJ** defined.

**Function Prototype:**

```
ST_INT u_gnl_ind_nvls (NAMELIST_REQ_INFO *req_info,
                       ST_CHAR **ptr,
                       ST_BOOLEAN *moreFollowsOut,
                       ST_INT maxNames);
```

**Parameters:**

| | |
|---|---|
| req_info | GetNameList request information. It is necessary to examine this structure to determine if there is a name to continue after. The structure **NAMELIST_REQ_INFO** is defined in **mms_pvmd.h**. |
| ptr | An array of pointers to named variable list name character strings. The user must fill in the array. |
| moreFollowsOut | Set ***moreFollowsOut** = **SD_TRUE** if not all named variable list names being manufactured by the application will fit in the NameList response. The maximum number of named variable list names that may be returned is supplied as the parameter **maxNames**. Set ***moreFollowsOut** = **SD_FALSE** when the function reports the last known manufactured named variable list name in the set. |
| maxNames | The maximum number of named variable list names that may be returned by the function per call. |

| | | |
|---|---|---|
| **Return Value:** | ST_INT | The number of named variable list names returned in the pointer table. 0 indicates that the function did not return any manufactured named variable list names. |

## u_gnl_ind_doms

**Usage:** When the application is making use of the manufactured object approach, it will also be necessary to provide the list of objects to be returned when a MMS GetNameList indication is received. The user must provide this function. Note that this function is only called when MVL is compiled with **USE_MANUFACTURED_OBJ** defined.

**Function Prototype:**

```
ST_INT u_gnl_ind_doms (NAMELIST_REQ_INFO *req_info,
                       ST_CHAR **ptr,
                       ST_BOOLEAN *moreFollowsOut,
                       ST_INT maxNames);
```

**Parameters:**

| | |
|---|---|
| `req_info` | GetNameList request information. It is necessary to examine this structure to determine if there is a name in the domain name space to continue after. The structure **NAMELIST_REQ_INFO** is defined in **mms_pvmd.h**. |
| `ptr` | An array of pointers to domain name character strings. The user must fill in the array. |
| `moreFollowsOut` | Set ***moreFollowsOut** = **SD_TRUE** if not all domain names being manufactured by the application will fit in the NameList response. The maximum number of domain names that may be returned is supplied as the parameter **maxNames**. Set ***moreFollowsOut** = **SD_FALSE** when the function reports the last known manufactured domain name in the set. |
| `maxNames` | The maximum number of domain names that may be returned by the function per call. |

| | | |
|---|---|---|
| **Return Value:** | `ST_INT` | The number of domain names returned in the pointer table. 0 indicates that the function did not return any manufactured domain names. |

## u_gnl_ind_jous

**Usage:** When the application is making use of the manufactured object approach, it will also be necessary to provide the list of objects to be returned when a MMS GetNameList indication is received. The user must provide this function. Note that this function is only called when MVL is compiled with **`USE_MANUFACTURED_OBJ`** defined.

**Function Prototype:**

```
ST_INT u_gnl_ind_jous (NAMELIST_REQ_INFO *req_info,
                       ST_CHAR **ptr,
                       ST_BOOLEAN *moreFollowsOut,
                       ST_INT maxNames);
```

**Parameters:**

`req_info` GetNameList request information. It is necessary to examine this structure to determine if there is a name to continue after. The structure **`NAMELIST_REQ_INFO`** is defined in **mms_pvmd.h**.

`ptr` An array of pointers to journal name character strings. The user must fill in the array.

`moreFollowsOut` Set **`*moreFollowsOut`** = **`SD_TRUE`** if not all journal names being manufactured by the application will fit in the NameList response. The maximum number of journal names that may be returned is supplied as the parameter **`maxNames`**. Set **`*moreFollowsOut`** = **`SD_FALSE`** when the function reports the last known manufactured journal name in the set.

`maxNames` The maximum number of journal names that may be returned by the function per call.

**Return Value:** `ST_INT` The number of journal names returned in the pointer table. 0 indicates that the function did not return any manufactured journal names.

# MVL Dynamic Object Management

MVL provides functions for adding and deleting MMS objects (e.g., Named Types, Named Variables, Named Variable Lists, Domains, and Journals) at runtime. These functions are useful for systems where the objects are not known at compile time.

See **mvl_defs.h** for the following set of constants that determine how many dynamic objects can be added. These constants are used in the source code generated by the MMS Object Foundry.

```
#define MVL_NUM_DYN_DOMS      10
#define MVL_NUM_DYN_VMD_VARS  10
#define MVL_NUM_DYN_VMD_NVLS  10
#define MVL_NUM_DYN_JOUS      10
#define MVL_NUM_DYN_DOM_VARS  10
#define MVL_NUM_DYN_DOM_NVLS  10
#define MVL_NUM_DYN_AA_VARS   10
#define MVL_NUM_DYN_AA_NVLS   10

/* MVL_UCA requires dynamic types to function */
#if defined(MVL_UCA)
#define MVLU_NUM_DYN_TYPES    100
#else
#define MVLU_NUM_DYN_TYPES    0
#endif
```

It is also possible to adjust the number of dynamic objects associated with the VMD by calling **`mvl_vmd_resize`**. The number of objects associated with a domain can be adjusted by calling **`mvl_dom_resize`**.

In applications where the MMS Object Foundry does not generate code to add any of a certain type of object, the constants found in **mvl_defs.h** are not used to generate the overhead necessary for dynamic object management. It is necessary to resize the number of objects using the **`mvl_vmd_resize`** and **`mvl_dom_resize`** functions. The maximum and current numbers of objects of any type are found by referencing the **`mvl_vmd`** control structure. **`mvl_vmd`** is also found in **mvl_defs.h**

## *MVL Dynamic Object Management Functions*

### mvl_dom_add

**Usage:** This function will add a Domain, allocate and modify the memory associated with its overhead, and insert it into the MMS-EASE *Lite* database. The value returned is a pointer to the new Domain.

**Function Prototype:**
```
MVL_DOM_CTRL *mvl_dom_add (ST_CHAR *name,
                           ST_INT max_num_var,
                           ST_INT max_num_nvl,
                           ST_INT max_num_jou,
                           ST_BOOLEAN copy_name);
```

**Parameters:**

| | |
|---|---|
| `name` | Name of the new Domain object. |
| `max_num_var` | Maximum number of Named Variables to allow in the Domain. |
| `max_num_nvl` | Maximum number of Named Variable Lists to allow in the Domain. |
| `max_num_jou` | Maximum number of Journals to allow in the Domain. |
| `copy_name` | Flag to indicate if the name should be copied to an allocated buffer. If **`SD_FALSE`**, the argument *name* must be the address of nonvolatile memory where the name is stored. |

| | | |
|---|---|---|
| **Return Value:** | `MVL_DOM_CTRL *` | Pointer to the new Domain object. **`NULL`** if operation failed. The structure **`MVL_DOM_CTRL`** is defined in **mvl_defs.h**. |

### mvl_dom_remove

**Usage:** This function will remove the **`MVL_DOM_CTRL`** structure from the MMS-EASE *Lite* database and deallocate and adjust the overhead associated with it. User callback function **`*u_mvl_dom_destroy`** is invoked in the process to allow the application a chance to deallocate any application specific resources associated with the Domain. Please see function **`u_mvl_dom_destroy`** for further details.

**Function Prototype:**
```
ST_RET mvl_dom_remove (ST_CHAR *dom_name);
```

**Parameters:**

| | |
|---|---|
| `dom_name` | This is the name of the domain to delete from the MMS-EASE *Lite* database. |

| | | | |
|---|---|---|---|
| **Return Value:** | `ST_RET` | `SD_SUCCESS` | No error code |
| | | <>0 | Error code |

## u_mvl_dom_destroy

**Usage:** When set by the application, this function pointer is invoked by the MMS-EASE *Lite* library during the process of removing a Domain from the MMS-EASE *Lite* database. The intent is to allow the application a chance to deallocate any resources associated with the Domain. By default, this function pointer is not set.

**Function Pointer Global Variable:**

```
extern ST_VOID (*u_mvl_dom_destroy)
                           (MVL_DOM_CTRL *dom);
```

**Parameters:**

| | |
|---|---|
| `dom` | A pointer to a domain control structure being freed from the MMS-EASE *Lite* database. |

**Return Value:** `ST_VOID`

## mvl_derive_new_type

**Usage:** This function will derive a new Named Type from a preexisting Named Type created by the MMS Object Foundry. Normally the new type is derived from a standard UCA type or a base class. The derivation is one which deletes type members from the base class so that the result models the data supported in a particular GOMSFE brick. New type members are not added with this function and if new members are required, the associated **.odf** file must be modified prior to running the MMS Object Foundry and calling this function. The function allocates and modifies the memory associated with Named Type overhead and inserts it into the MMS-EASE *Lite* database. The **`typeIdOut`** parameter contains the newly created Named Type ID. A user-supplied function called **`u_mvl_rt_element_supported`** is invoked for each member of the base class. See also the related function **`u_mvl_rt_element_supported`**.

**Note:** Special code is needed to release the overhead associated with type IDs created when calling **`mvl_derive_new_type`**. Do not call **`mvl_type_id_destroy`** with type IDs returned from this function. The results will be unpredictable.

**Function Prototype:**

```
ST_RET  mvl_derive_new_type (ST_CHAR *base_name,
                             ST_INT typeIdIn,
                             ST_INT *typeIdOut,
                             ST_CHAR *handle);
```

**Parameters:**

| | |
|---|---|
| `base_name` | This is the string that will be prefixed onto the derived named type object. |
| `typeIdIn` | This is the type ID of the existing Named Type that is used as the base class for derivation. |
| `typeIdOut` | This output parameter is the type ID of the derived type. It may be used to add a Named Variable. See associated function **`mvl_var_add`**. |
| `handle` | This is a pointer to any user defined string or object that the application may need to see when examining individual type members in **`u_mvl_rt_element_supported`**. |

**Return Value:**

| | | |
|---|---|---|
| `ST_RET` | `SD_SUCCESS` | No error code |
| | <0 | Error code |

## mvl_type_id_create

**Usage:** This function will parse an ASN.1 encoded Named Type and add the resulting runtime type. The function allocates and modifies the memory associated with Named Type overhead and inserts it into the MMS-EASE *Lite* database. The return value can be used to perform variable access. See associated **mvla_getvar**, **mvla_read_variables**, and **mvla_write_variables** functions.

**Function Prototype:**

```
ST_RET mvl_type_id_create (ST_CHAR *type_name,
                           ST_UCHAR *asn1_data,
                           ST_UINT asn1_len);
```

**Parameters:**

`type_name` Name of this type. Stored with type definition. May be used later to find this type ID using **mvl_typename_to_typeid**. Use NULL if name is not needed.

`asn1_data` This is the ASN.1 encoded type specification typically seen in an MMS GetVariableAccessAttributes-Response. See **GETVAR_RESP_INFO** for more information.

`asn1_len` This is the length of the ASN.1 encoded type specification.

| | | | |
|---|---|---|---|
| **Return Value:** | ST_RET | -1 | Type creation failed |
| | | >=0 | Type creation succeeded and this is the new type ID. |

## mvl_type_id_create_from_tdl

**Usage:** This function creates a type definition from TDL.

**Function Prototype:**

```
ST_RET mvl_type_id_create_from_tdl (ST_CHAR *type_name,
                                    ST_CHAR *tdl);
```

**Parameters:**

`type_name` Name of this type. Stored with type definition. May be used later to find this type ID using `mvl_typename_to_typeid.` Use NULL if name is not needed.

`tdl` TDL string to define new type.

| | | | |
|---|---|---|---|
| **Return Value:** | ST_RET | -1 | Type creation failed |
| | | >=0 | Type creation succeeded and this is the new type ID. |

## mvl_type_id_destroy

**Usage:** This function will release the overhead associated with a typeID, which was created by calling **`mvl_type_id_create`**.

**Function Prototype:** `ST_VOID mvl_type_id_destroy (ST_INT TypeId);`

**Parameters:**

| | |
|---|---|
| `TypeId` | This is a typeID created by a call to **`mvl_type_id_create`**. |

**Return Value:** `ST_VOID`

## mvl_type_ctrl_find

**Usage:** This function will find the type ctrl structure corresponding to the type id.

**Function Prototype:** `MVL_TYPE_CTRL *mvl_type_ctrl_find (ST_INT TypeId);`

**Parameters:**

| | |
|---|---|
| `TypeId` | This is a typeID created by a call to **`mvl_type_id_create`**. |

**Return Value:** `MVL_TYPE_CTRL` pointer or **`NULL`** on error.

## mvl_var_add

**Usage:** This function will add a Named Variable, allocate and modify the memory associated with its overhead, and insert it into the MMS-EASE *Lite* database. The value returned is a pointer to the new Named Variable.

**Function Prototype:**

```
MVL_VAR_ASSOC *mvl_var_add (OBJECT_NAME *obj,
                            MVL_NET_INFO *net_info,
                            ST_INT type_id,
                            ST_VOID *data,
                            MVL_VAR_PROC *proc,
                            ST_BOOLEAN copy_name);
```

**Parameters:**

`obj` The MMS Object Name of the Named Variable object to add. The structure **`OBJECT_NAME`** is defined in **mms_mp.h**.

`net_info` Network connection information required when the scope of the named variable is **`AA_SPEC`**. This may be **`NULL`** when the scope is **`VMD_SPEC`** or **`DOM_SPEC`**.

`type_id` This is the MMS Object Foundry generated TypeID that represents the data type associated with the NamedVariable.

`data` This is a pointer to where the variable resides in memory.

`proc` This is an optional pointer to a structure of type **`MVL_VAR_PROC`**. It associates specific functions to be called when the NamedVariable is read or written.

`copy_name` Flag to indicate if the name should be copied to an allocated buffer. If **`SD_FALSE`**, the argument *name* must be the address of nonvolatile memory where the name is stored.

**Return Value:** `MVL_VAR_ASSOC *` Pointer to the new Named Variable object. **`NULL`** if operation failed. The structure **`MVL_VAR_ASSOC`** is defined in **mvl_defs.h**.

## mvl_var_remove

**Usage:** This function will remove the **`MVL_VAR_ASSOC`** structure from the MMS-EASE *Lite* database and deallocate and adjust the overhead associated with it. User callback function **`*u_mvl_var_destroy`** is invoked in the process to allow the application a chance to deallocate any application specific resources associated with the Named Variable. Please see function **`u_mvl_var_destroy`** for further details.

**Function Prototype:**

```
ST_RET mvl_var_remove (OBJECT_NAME *obj,
                       MVL_NET_INFO *net_info);
```

**Parameters:**

| | |
|---|---|
| `obj` | The MMS Object Name of the Named Variable object to remove. The structure **`OBJECT_NAME`** is defined in **mms_mp.h**. |
| `net_info` | Network connection information required when the scope of the named variable is **`AA_SPEC`**. This may be **`NULL`** when the scope is **`VMD_SPEC`** or **`DOM_SPEC`**. |

| | | | |
|---|---|---|---|
| **Return Value:** | `ST_RET` | `SD_SUCCESS` | No error code |
| | | <>0 | Error code |

## u_mvl_var_destroy

**Usage:** When set by the application, this function pointer is invoked by the MMS-EASE *Lite* library during the process of removing a Variable from the MMS-EASE *Lite* database. The intent is to allow the application a chance to deallocate any resources associated with the Variable. By default, this function pointer is not set.

**Function Pointer Global Variable:**

```
extern ST_VOID *u_mvl_var_destroy)
                (MVL_VAR_ASSOC *va);
```

**Parameters:**

| | |
|---|---|
| `va` | A pointer to a variable association control structure being freed from the MMS-EASE *Lite* database. |

**Return Value:** `ST_VOID`

## mvl_nvl_add

**Usage:** This function will add a Named Variable List, allocate and modify the memory associated with its overhead, and insert it into the MMS-EASE *Lite* database. The value returned is a pointer to the new Named Variable List.

**Function Prototype:**

```
MVL_NVLIST_CTRL *mvl_nvl_add (OBJECT_NAME *obj,
                              MVL_NET_INFO *net_info,
                              ST_INT num_var,
                              OBJECT_NAME *var_obj,
                              ST_BOOLEAN copy_name);
```

**Parameters:**

| | |
|---|---|
| `obj` | The MMS Object Name of the Named Variable List object to add. The structure **`OBJECT_NAME`** is defined in **mms_mp.h**. |
| `net_info` | Network connection information required when the scope of the named variable is **`AA_SPEC`**. This may be **`NULL`** when the scope is **`VMD_SPEC`** or **`DOM_SPEC`**. The structure **`MVL_NET_INFO`** is defined in **mvl_defs.h** |
| `num_var` | This is the number of variables in the **`var_obj`** array. |
| `var_obj` | MMS Object Name array of Named Variables included in the Named Variable List. The structure **`OBJECT_NAME`** is defined in **mms_mp.h**. |
| `copy_name` | Flag to indicate if the name should be copied to an allocated buffer. If **`SD_FALSE`**, the argument *name* must be the address of nonvolatile memory where the name is stored. |

| | | |
|---|---|---|
| **Return Value:** | `MVL_NVLIST_CTRL *` | Pointer to the new Named Variable List object. **`NULL`** if the operation failed. The structure **`MVL_NVLIST_CTRL`** is defined in **mvl_defs.h**. |

## mvl_nvl_remove

**Usage:** This function will remove the **`MVL_NVLIST_CTRL`** structure from the MMS-EASE *Lite* database and deallocate and adjust the overhead associated with it. User callback function **`*u_mvl_nvl_destroy`** is invoked in the process to allow the application a chance to deallocate any application specific resources associated with the Named Variable List. Please see function **`u_mvl_nvl_destroy`** for further details.

**Function Prototype:**

```
ST_RET mvl_nvl_remove (OBJECT_NAME *obj,
                       MVL_NET_INFO *net_info);
```

**Parameters:**

`obj` The MMS Object Name of the Named Variable List object to remove. The structure **`OBJECT_NAME`** is defined in **mms_mp.h**.

`net_info` This is required when the scope of the Named Variable List is **`AA_SPEC`**. This may be **`NULL`** when the scope is **`VMD_SPEC`** or **`DOM_SPEC`**. The structure **`MVL_NET_INFO`** is defined in **mvl_defs.h**.

| | | | |
|---|---|---|---|
| **Return Value:** | `ST_RET` | `SD_SUCCESS` | No error code |
| | | <>0 | Error code |

## u_mvl_nvl_destroy

**Usage:** When set by the application, this function pointer is invoked by the MMS-EASE *Lite* library during the process of removing a Named Variable List from the MMS-EASE *Lite* database. The intent is to allow the application a chance to deallocate any resources associated with the Variable. By default, this function pointer is not set.

**Function Pointer Global Variable:**

```
extern ST_VOID (*u_mvl_nvl_destroy)
               (MVL_NVLIST_CTRL *nvl);
```

**Parameters:**

`nvl` A pointer to a Named Variable List control structure being freed from the MMS-EASE *Lite* database.

**Return Value:** `ST_VOID`

## mvl_jou_add

**Usage:** This function will add a Journal, allocate and modify the memory associated with its overhead, and insert it into the MMS-EASE *Lite* database. The value returned is a pointer to the new Journal.

**Function Prototype:**

```
MVL_JOURNAL_CTRL *mvl_jou_add (OBJECT_NAME *obj,
                               MVL_NET_INFO *net_info,
                               ST_BOOLEAN copy_name);
```

**Parameters:**

`obj` The MMS Object Name of the Journal object to add. The structure **`OBJECT_NAME`** is defined in **mms_mp.h**.

`net_info` This is required when the scope of the Journal is **`AA_SPEC`**. This may be **`NULL`** when the scope is **`VMD_SPEC`** or **`DOM_SPEC`**. The structure **`MVL_NET_INFO`** is defined in **mvl_defs.h**.

`copy_name` Flag to indicate if the name should be copied to an allocated buffer. If **`SD_FALSE`**, the argument *name* must be the address of nonvolatile memory where the name is stored.

| | | |
|---|---|---|
| **Return Value:** | `MVL_JOURNAL_CTRL *` | Pointer to the new Journal object. **`NULL`** if the operation failed. The structure **`MVL_JOURNAL_CTRL`** is defined in **mvl_defs.h**. |

## mvl_jou_remove

**Usage:** This function will remove the **`MVL_JOURNAL_CTRL`** structure from the MMS-EASE *Lite* database and deallocate and adjust the overhead associated with it. User callback function **`*u_mvl_jou_destroy`** is invoked in the process to allow the application a chance to deallocate any application specific resources associated with the Journal. Please see **`u_mvl_jou_destroy`** for further details.

**Function Prototype:**

```
ST_RET mvl_jou_remove (OBJECT_NAME *obj,
                       MVL_NET_INFO *net_info);
```

**Parameters:**

`obj` The MMS Object Name of the Journal object to remove. The structure **`OBJECT_NAME`** is defined in **mms_mp.h**.

`net_info` This is required when the scope of the Journal is **`AA_SPEC`**. This may be **`NULL`** when the scope is **`VMD_SPEC`** or **`DOM_SPEC`**. The structure **`MVL_NET_INFO`** is defined in **mvl_defs.h**.

| | | | |
|---|---|---|---|
| **Return Value:** | `ST_RET` | `SD_SUCCESS` | No error code |
| | | <>0 | Error code |

## u_mvl_jou_destroy

**Usage:** When set by the application, this function pointer is invoked by the MMS-EASE *Lite* library during the process of removing a Journal from the MMS-EASE *Lite* database. The intent is to allow the application a chance to deallocate any resources associated with the Journal. By default, this function pointer is not set.

**Function Pointer Global Variable:**

```
extern ST_VOID (*u_mvl_jou_destroy)
                (MVL_JOURNAL_CTRL *jou);
```

**Parameters:**

| | |
|---|---|
| `jou` | A pointer to a Journal control structure being freed from the MMS-EASE *Lite* database. |

**Return Value:** `ST_VOID`

## mvl_vmd_resize

**Usage:** This function sets up the overhead for the maximum number of objects that may be added to the VMD specific portion of the MMS-EASE *Lite* database. The parameter values may be increased or decreased as memory requirements permit. Note that these values must take into consideration the total number of objects the MMS Object Foundry (static)added and those that were dynamically added. The current number of each type of object may be found by examining members of the global variable **`mvl_vmd`**. The **`MVL_VMD_CTRL`** structure is defined in **mvl_defs.h**.

**Function Prototype:**

```
ST_VOID mvl_vmd_resize (ST_INT max_dom,
                        ST_INT max_var,
                        ST_INT max_nvl,
                        ST_INT max_jou);
```

**Parameters:**

| | |
|---|---|
| `max_dom` | The new maximum number of Domain objects associated with the VMD. |
| `max_var` | The new maximum number of Named Variable objects associated with the VMD. |
| `max_nvl` | The new maximum number of Named Variable List objects associated with the VMD. |
| `max_jou` | The new maximum number of Journal objects associated with the VMD. |

**Return Value:** `ST_VOID`

## mvl_dom_resize

**Usage:** This function sets up the overhead for the maximum number of objects that may be added to a Domain specific portion of the MMS-EASE *Lite* database. The parameter values may be increased or decreased as memory requirements permit. Note that these values must take into consideration the total number of objects the MMS Object Foundry (static)added and those that were dynamically added. The current number of each type of object may be found by examining the **MVL_DOM_CTRL** structure. To find the **MVL_DOM_CTRL** call the function **mvl_find_dom**.

**Function Prototype:**

```
ST_VOID mvl_dom_resize (MVL_DOM_CTRL *dom,
                        ST_INT max_var,
                        ST_INT max_nvl,
                        ST_INT max_jou);
```

**Parameters:**

| | |
|---|---|
| dom | A pointer to a MVL control structure representing the Domain. The **MVL_DOM_CTRL** structure is defined in **mvl_defs.h**. |
| max_var | The new maximum number of Named Variable objects associated with the Domain. |
| max_nvl | The new maximum number of Named Variable List objects associated with the Domain. |
| max_jou | The new maximum number of Journal objects associated with the Domain. |

**Return Value:** ST_VOID

# MMS Object Control

## Configured and Manufactured MMS Server Objects

Server objects are those MMS variables, variable lists, and domains that are visible to MMS Clients. MVL supports both configured and manufactured variables and variable lists.

Configured objects are those that are configured in the **`MVL_VMD_CTRL`** data structure and which MVL can handle transparently if desired. This is the simplest way to handle variable access, and most applications should use this approach.

Manufactured objects are those that are NOT configured in the **`MVL_VMD_CTRL`** data structure, and so the user must resolve the MMS object to local mapping dynamically. This takes a bit more work than using configured objects, but can be useful under some conditions. When using manufactured objects, the user must also handle the MMS GetNameList indications directly.

It is possible to mix configured and manufactured objects, and the MVL server sample application does just that. The define, **`USE_MANUFACTURED_OBJS`**, is used to isolate the sections of code required for manufactured objects in the sample server application.

### *MVL Type Handling*

In MVL, types are referenced by an integer index into a table of **`MVL_TYPE_CTRL`** elements. This table is created off-line by using MMS Object Foundry with an Object Definition File containing TDL strings. MMS Object Foundry creates a C file that contains all code required to create the type table.

# MMS Object Scope

The following figure provides a review of the MMS object scopes. The MMS specifications will also provide useful information in this regard. Please refer to *Prerequisites* on page 3 for additional information MMS specifications.

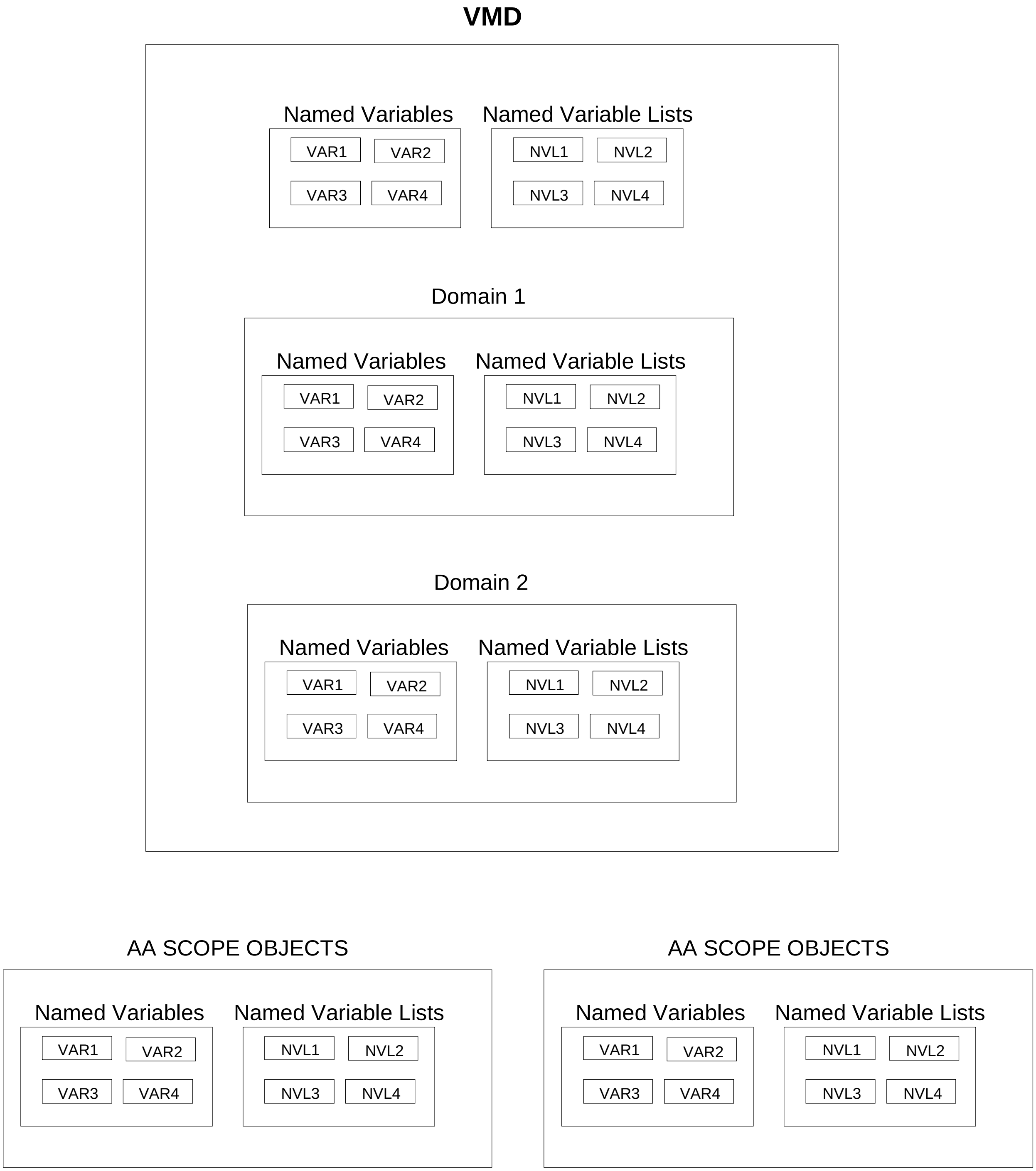


Figure 1: MMS Object Scope Overview

# The MVL VMD Control Data Structure

MMS-EASE *Lite* provides a MMS VMD model that is used to associate MMS (Named Variables, Named Variable Lists, and Domains) to local objects; implemented by the **`MVL_VMD_CTRL`** data.

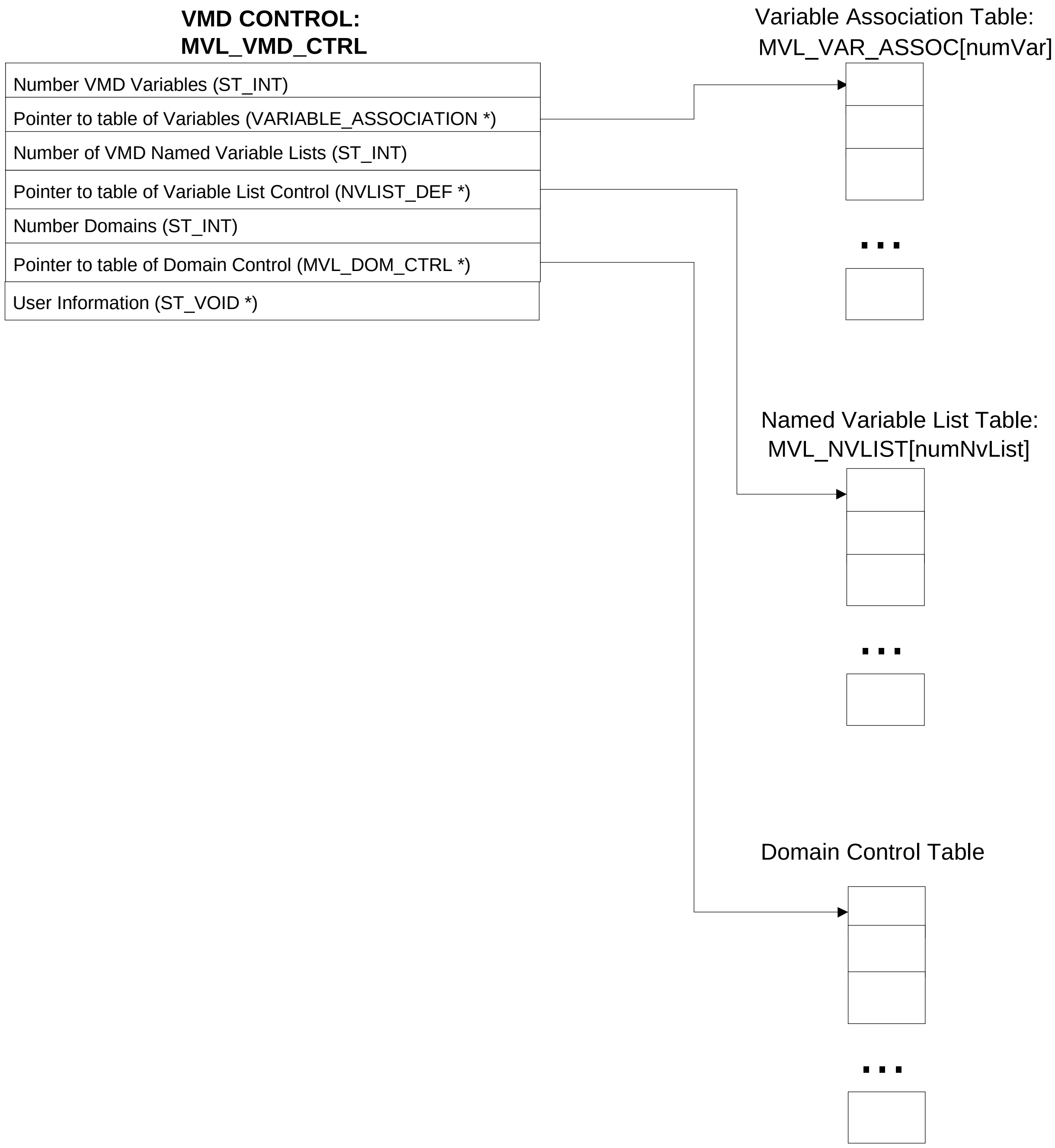

VMD CONTROL:
MVL_VMD_CTRL
Number VMD Variables (ST_INT)
Pointer to table of Variables (VARIABLE_ASSOCIATION *)
Number of VMD Named Variable Lists (ST_INT)
Pointer to table of Variable List Control (NVLIST_DEF *)
Number Domains (ST_INT)
Pointer to table of Domain Control (MVL_DOM_CTRL *)
User Information (ST_VOID *)
Variable Association Table:
MVL_VAR_ASSOC[numVar]
Named Variable List Table:
MVL_NVLIST[numNvList]
Domain Control Table

Figure 2: MVL VMD Control Data Structure

```
typedef struct mvl_vmd_ctrl
  {
  ST_INT max_num_var_assoc;
  ST_INT num_var_assoc;
  MVL_VAR_ASSOC **var_assoc_tbl;

#if defined (MVL_DESCR_SUPP)
  ST_INT num_descr_addr;
  MVL_DESCR_ADDR_ASSOC *descr_addr_assoc_tbl;
#endif

  ST_INT max_num_nvlist;
  ST_INT num_nvlist;
  MVL_NVLIST_CTRL **nvlist_tbl;

  ST_INT max_num_dom;
  ST_INT num_dom;
  MVL_DOM_CTRL **dom_tbl;

  ST_INT max_num_jou;
  ST_INT num_jou;
  MVL_JOURNAL_CTRL **jou_tbl;

  ST_BOOLEAN foundry_objects; /* Flag for internal use */

  ST_VOID *user_info;         /* MVL user can use this for 'whatever'   */
  } MVL_VMD_CTRL;
```

## The MVL Domain Control Data Structure

The MMS-EASE *Lite* VMD model supports Domain scope objects (Named Variables and Named Variable Lists). The **`MVL_DOM_CTRL`** data structure references arrays of these objects and MVL provides transparent access to client applications.

```
typedef struct mvl_dom_ctrl
  {
  ST_CHAR *name;

  ST_INT max_num_var_assoc;
  ST_INT num_var_assoc;
  MVL_VAR_ASSOC **var_assoc_tbl;

  ST_INT max_num_nvlist;
  ST_INT num_nvlist;
  MVL_NVLIST_CTRL **nvlist_tbl;

  ST_INT max_num_jou;
  ST_INT num_jou;
  MVL_JOURNAL_CTRL **jou_tbl;

  GETDOM_RESP_INFO  *get_dom_resp_info; /* Optional               */

  ST_BOOLEAN foundry_objects; /* Flag for internal use */
  ST_VOID *user_info;         /* MVL user can use this for 'whatever'*/
  } MVL_DOM_CTRL;
```

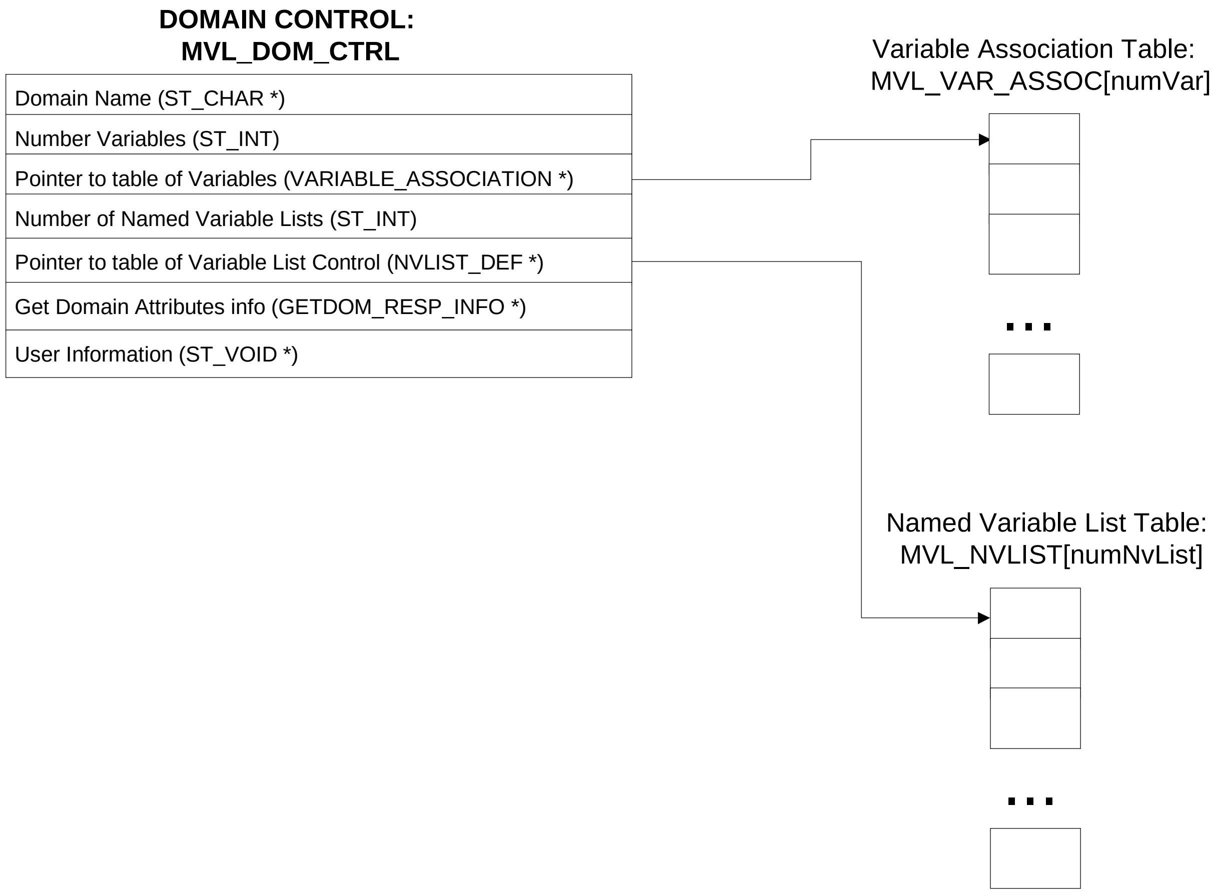

DOMAIN CONTROL:
MVL_DOM_CTRL
Domain Name (ST_CHAR *)
Number Variables (ST_INT)
Pointer to table of Variables (VARIABLE_ASSOCIATION *)
Number of Named Variable Lists (ST_INT)
Pointer to table of Variable List Control (NVLIST_DEF *)
Get Domain Attributes info (GETDOM_RESP_INFO *)
User Information (ST_VOID *)
Variable Association Table:
MVL_VAR_ASSOC[numVar]
...
Named Variable List Table:
MVL_NVLIST[numNvList]
...

Figure 3: MVL Domain Control Data Structure

# The MVL AA Control Data Structure

The MMS-EASE *Lite* Application Association object control structure is used to expose local objects (Named Variables and Named Variable Lists) as MMS AA scope objects. AA scope objects are typically used to give each client application an independent copy of named objects. For instance, AA scope is useful to allow a MMS client to select reports to be generated by the server. In MMS-EASE *Lite*, the **MVL_AA_OBJ_CTRL** data structure can be attached to each MMS connection.

```
typedef struct mvl_aa_obj_ctrl
  {
  ST_INT max_num_var_assoc;
  ST_INT num_var_assoc;
  MVL_VAR_ASSOC **var_assoc_tbl;

  ST_INT max_num_nvlist;
  ST_INT num_nvlist;
  MVL_NVLIST_CTRL **nvlist_tbl;

  ST_INT max_num_jou;
  ST_INT num_jou;
  MVL_JOURNAL_CTRL **jou_tbl;
ST_BOOLEAN foundry_objects;    /* Flag for internal use */
  ST_VOID *user_info;          /* MVL user can use this for 'whatever'*/
  } MVL_AA_OBJ_CTRL;
```

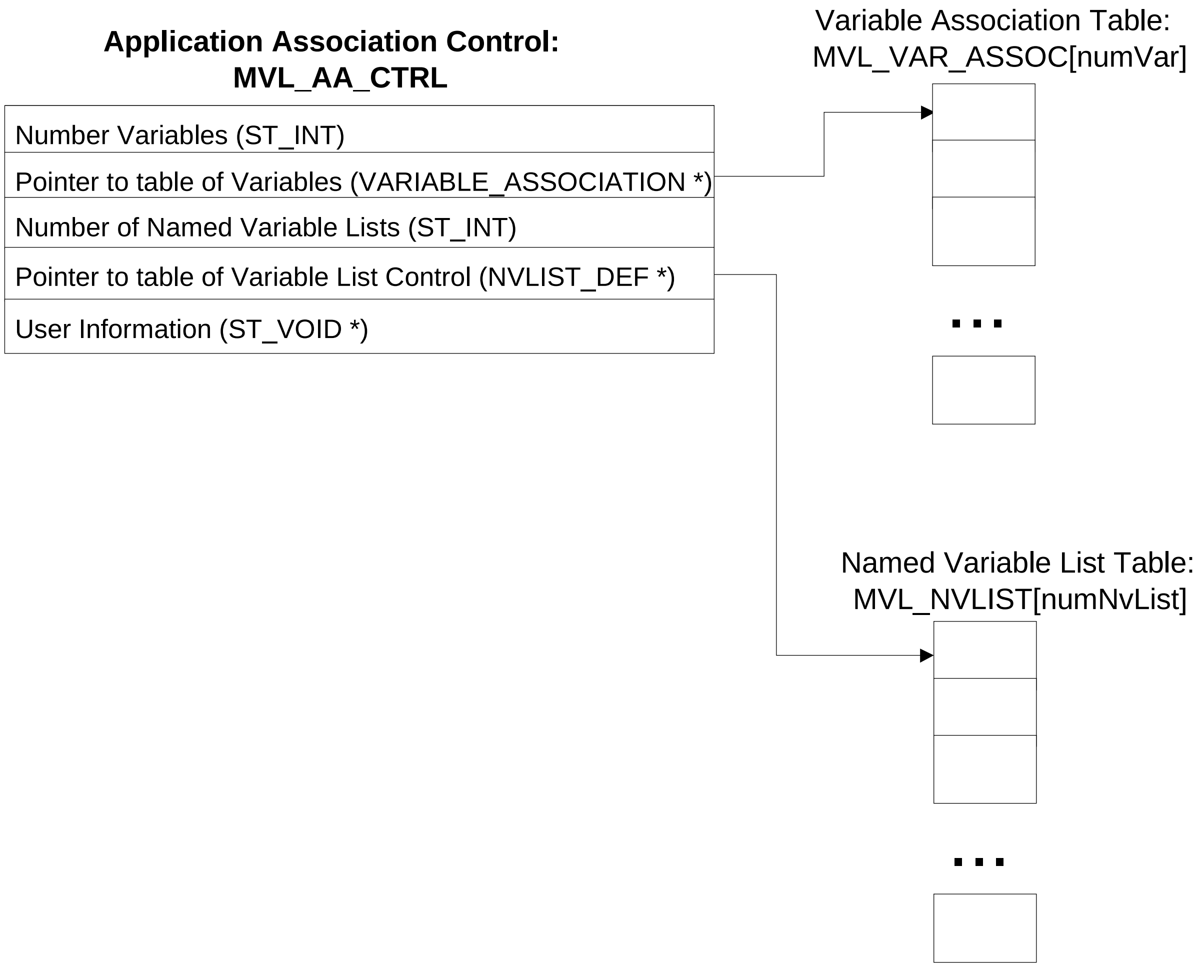

Application Association Control:
MVL_AA_CTRL
Number Variables (ST_INT)
Pointer to table of Variables (VARIABLE_ASSOCIATION *)
Number of Named Variable Lists (ST_INT)
Pointer to table of Variable List Control (NVLIST_DEF *)
User Information (ST_VOID *)
Variable Association Table:
MVL_VAR_ASSOC[numVar]
Named Variable List Table:
MVL_NVLIST[numNvList]

Figure 4: MVL AA Control Data Structure

# The MVL Named Variable List Data Structure

A MMS-EASE *Lite* Named Variable List control structure simply references a set of LITE Variable Association control structures. Note that in UCA, a DataSet is implemented by a MMS Named Variable List.

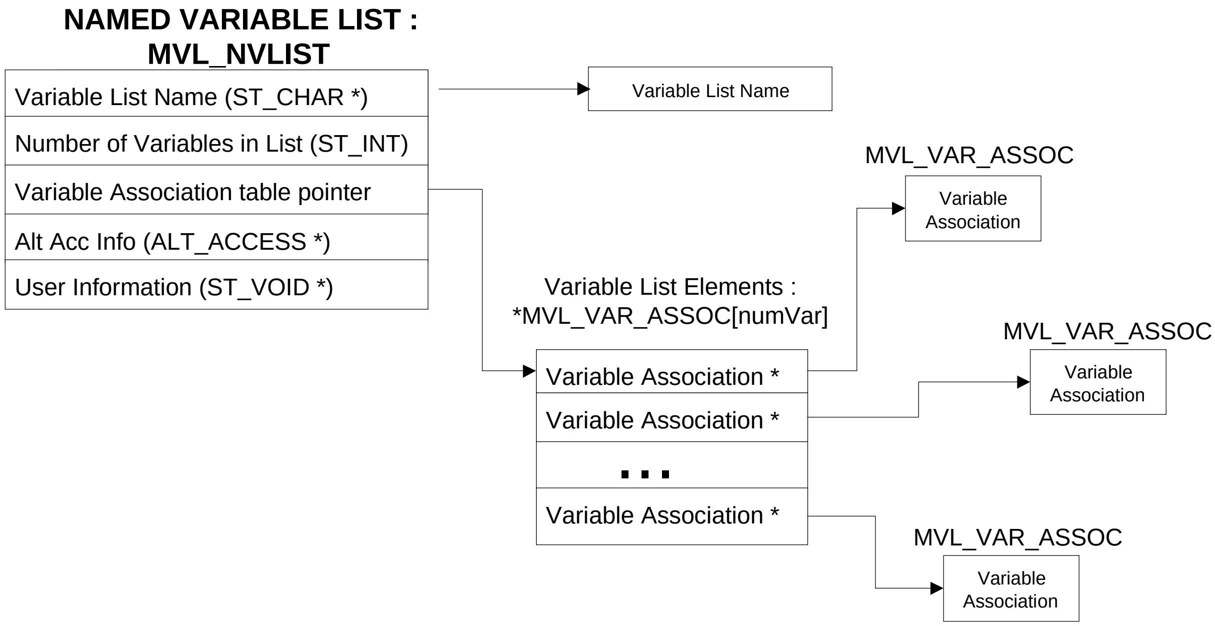


Figure 5: MVL Named Variable List Data Structure

```
typedef struct mvl_nvlist_ctrl
 {
 ST_CHAR *name;                 /* name of the named variable list */
 ST_INT num_of_entries;         /* number of variables in the list */
 MVL_VAR_ASSOC **entries;
 MVL_SCOPE nvl_scope;          /* scope of this NVL                */
 MVL_SCOPE *va_scope;
 ALT_ACCESS **altAcc;          /* Alternate Access array for var's */
 ST_BOOLEAN mms_deletable;
 ST_VOID *user_info;           /* MVL user can use this for 'whatever'*/
 } MVL_NVLIST_CTRL;
```

# MVL MMS Server Facilities

MVL provides a subsystem that performs the actual services requested by the remote device for many MMS services. These functions require only appropriate configuration, and no other user code is involved unless required by the application. MVL provides a straightforward mechanism by which named variable access can be mapped to local variables. MVL supports VMD, Domain, and AA scopes. Please refer to the MVL sample server source code (**server.c**) for more details.

# Synchronous vs. Asynchronous Response - Indication Control

The user application can respond to any MMS indication either synchronously or asynchronously, as needed. A user indication function is called from **`mvl_comm_serve`** whenever a MMS indication is received. A pointer to an indication control structure (**`MVL_IND_PEND`**) is passed to each indication function. It contains both request and response data for the indication. To send the response for an indication, the same pointer that is passed to the indication function must be passed to the response function. If the indication can be processed immediately (i.e., synchronously), the indication function should set the response data and then call the response function (as is done in most of the sample server source code). However, some applications are not able to respond immediately to some types of indications. For instance, the application may need to acquire read data via a serial link before it is able to respond. Other applications may not want responses sent automatically. In these cases, the application can simply save the pointer to the indication control structure (**`MVL_IND_PEND`**), and call the response function sometime later when the response is ready (i.e., asynchronously).

## *Error Response Function*

The following function may be used to send an error response for any indication.

### mplas_err_resp

**Usage:** This function is used to send an error response (result(-)) PDU for any confirmed service request except for Cancel, Conclude, and Initiate.

**Function Prototype:**

```
ST_RET mplas_err_resp (MVL_IND_PEND *ind_pend,
                       ST_INT16 err_class,
                       ST_INT16 code);
```

**Parameters:**

| | |
|---|---|
| `ind_pend` | This is the same parameter that is passed to all user-defined Indication functions. |
| `err_class` | This integer contains the particular class of the error per ISO 9506. |
| `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. |

| | | | |
|---|---|---|---|
| **Return Value:** | `ST_RET` | `SD_SUCCESS` | No Error. |
| | | `!= SD_SUCCESS` | Error (error response not sent). |

# MVL Indication Control Structure

The following is the MVL Indication Control Structure:

```
typedef struct mvl_ind_pend
  {
  DBL_LNK l;
  MVL_COMM_EVENT *event;
  ST_INT op;                  /* MMS Opcode (MMSOP_READ, etc.)    */
  union
    {
    MVLAS_READ_CTRL  rd;
    MVLAS_WRITE_CTRL wr;
    MVLAS_IDENT_CTRL ident;
    MVLAS_STATUS_CTRL status;
    MVLAS_NAMELIST_CTRL namelist;
    MVLAS_GETVAR_CTRL getvar;
    MVLAS_GETDOM_CTRL getdom;
    MVLAS_FOPEN_CTRL  fopen;
    MVLAS_FREAD_CTRL  fread;
    MVLAS_FCLOSE_CTRL fclose;
    MVLAS_FDIR_CTRL fdir;
    MVLAS_OBTFILE_CTRL obtfile;
    MVLAS_FDELETE_CTRL fdelete;
    MVLAS_FRENAME_CTRL frename;
    MVLAS_DEFVLIST_CTRL defvlist;
    MVLAS_GETVLIST_CTRL getvlist;
    MVLAS_DELVLIST_CTRL delvlist;
    MVLAS_JINIT_CTRL jinit;
    MVLAS_JREAD_CTRL jread;
    MVLAS_JSTAT_CTRL jstat;
    MVLAS_GETCL_CTRL getcl;
    } u;
  ST_VOID *usr_ind_ctrl;
  ST_VOID *usr;               /* For user to use as they see fit  */
  } MVL_IND_PEND;
```

# Status Service

This service is used to allow a client to determine the general condition or status of a server node.

### *Status Data Structures*

The following is the Status Indication Control Structure:

```
typedef struct
  {
  STATUS_REQ_INFO  *req_info;
  STATUS_RESP_INFO *resp_info;
  } MVLAS_STATUS_CTRL;
```

### STATUS_REQ_INFO

The client to issue the Status request uses the operation-specific structure described below. The server receives it when a Status indication 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).

### STATUS_RESP_INFO

The server in issuing the Status response uses the operation specific data structure described below. The client receives it when a Status confirm 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.

`local_detail` This implementation-specific bitstring contains additional data about the status of the VMD. It is defined by the particular VMD and is an array of 16 bytes (i.e., 128 bits).

## *Status Functions*

### u_mvl_status_ind

**Usage:** This is a user-defined function called when a Status indication is received. The user must first examine the request parameters contained in the **`MVL_IND_PEND`** structure. Second, do whatever is necessary to process the request. Third, fill in the response parameters in the **`MVL_IND_PEND`** structure. And finally, call **`mplas_status_resp`** to send the response (or **`mplas_err_resp`** to send an error response).

**Function Prototype:** `ST_VOID u_mvl_status_ind (MVL_IND_PEND *ind_pend);`

**Parameters:**

`ind_pend` This is the same parameter that is passed to all user defined Indication functions. It contains a union of request and/or response parameters that is used for several different services. The request parameters are set by MVL before calling this function. The user must set the response parameters before calling the response function (i.e., **`mplas_status_resp`**). The parameters to be used for this service are as follows:

**Request parameters:**

| | |
|---|---|
| `ind_pend->u.status.req_info` | See **`STATUS_REQ_INFO`** for more information. |

**Response parameters:**

| | |
|---|---|
| `ind_pend->u.status.resp_info` | See **`STATUS_RESP_INFO`** for more information. |

**Return Value:** `ST_VOID`

## mplas_status_resp

**Usage:** This function encodes and sends the Response for a previously received Status indication. The Response parameters in the **`MVL_IND_PEND`** structure must be filled in before this function is called. The (**`MVL_IND_PEND *`**) argument passed to this function must be the same as the (**`MVL_IND_PEND *`**) argument passed to the user defined function **`u_mvl_status_ind`** when the indication was received.

**Function Prototype:** `ST_VOID mplas_status_resp (MVL_IND_PEND *ind_pend);`

**Parameters:**

`ind_pend` This is the same parameter that is passed to all user defined Indication functions. It contains a union of request and/or response parameters that is used for several different services. The request parameters are set by MVL before calling this function. The user must set the response parameters before calling the response function (i.e., **`mplas_status_resp`**). The parameters to be used for this service are as follows:

**Request parameters:**

| | |
|---|---|
| `ind_pend->u.status.req_info` | See **`STATUS_REQ_INFO`** for more information. |

**Response parameters:**

| | |
|---|---|
| `ind_pend->u.status.resp_info` | See **`STATUS_RESP_INFO`** for more information. |

**Return Value:** `ST_VOID`

# Identify Service

This service is used to obtain identifying information such as a vendor name and model number from a responding node.

## *Identity Data Structures*

The following is the Identify Indication Control Structure:

```
typedef struct
  {
  IDENT_RESP_INFO *resp_info;
  } MVLAS_IDENT_CTRL;
```

### IDENT_RESP_INFO

The server in issuing an Identify response uses the operation specific structure described below. The client receives it when an Identify confirm 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 syntaxes. |

***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 *))));
```

## *Identify Functions*

## u_mvl_ident_ind

**Usage:** This is a user-defined function called when an Identify Indication is received. The user must first examine the request parameters contained in the **`MVL_IND_PEND`** structure. Second, do whatever is necessary to process the request. Third, fill in the response parameters in the **`MVL_IND_PEND`** structure. And finally, call **`mplas_ident_resp`** to send the response (or **`mplas_err_resp`** to send an error response).

**Function Prototype:** `ST_VOID u_mvl_ident_ind (MVL_IND_PEND *ind_pend);`

**Parameters:**

`ind_pend` This is the same parameter that is passed to all user defined Indication functions. It contains a union of request and/or response parameters that is used for several different services. The request parameters are set by MVL before calling this function. The user must set the response parameters before calling the response function (i.e., **`mplas_ident_resp`**). The parameters to be used for this service are as follows:

**Request parameters:**

NONE

**Response parameters:**

| | |
|---|---|
| `ind_pend->u.ident.resp_info` | See **`IDENT_RESP_INFO`** for more information. |

**Return Value:** `ST_VOID`

## mplas_ident_resp

**Usage:** This function encodes and sends the Response for a previously received Identify indication. The Response parameters in the **`MVL_IND_PEND`** structure must be filled in before this function is called. The (**`MVL_IND_PEND *`**) argument passed to this function must be the same as the (**`MVL_IND_PEND *`**) argument passed to the user defined function **`u_mvl_ident_ind`** when the indication was received.

**Function Prototype:** `ST_VOID mplas_ident_resp (MVL_IND_PEND *ind_pend);`

**Parameters:**

`ind_pend` This is the same parameter that is passed to all user defined Indication functions. It contains a union of request and/or response parameters that is used for several different services. The user must set the response parameters before calling the response function (i.e., **`mplas_ident_resp`**). The parameters to be used for this service are as follows:

**Request parameters:**

NONE

**Response parameters:**

| | |
|---|---|
| `ind_pend->u.ident.resp_info` | See **`IDENT_RESP_INFO`** for more information. |

**Return Value:** `ST_VOID`

# 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.

## *GetNameList Data Structures*

The following is the GetNameList Indication Control Structure:

```
typedef struct
  {
  NAMELIST_REQ_INFO *req_info;
  NAMELIST_RESP_INFO *resp_info;
  } MVLAS_NAMELIST_CTRL;
```

### NAMELIST_REQ_INFO

The client in issuing a GetNameList request uses this operation specific structure described below. The server receives it when a GetNameList indication 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.

### NAMELIST_RESP_INFO

This operation specific structure described below is used by the server in issuing a GetNameList response. It is received by the client when a GetNameList confirm 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 after this response.

**`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. The structure and name pointers must be allocated in a single call to **`chk_malloc`** of size: **`(sizeof(NAMELIST_RESP_INFO) + num_names * sizeof(ST_CHAR *))`**.

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 *))));
```

### *GetNameList Functions*

## u_mvl_namelist_ind

**Usage:** This is a user-defined function called when a GetNameList indication is received. The user must examine the request parameters contained in the **`MVL_IND_PEND`** structure, do whatever is necessary to process the request, and then call **`mplas_namelist_resp`** or **`mvlas_namelist_resp`** to send the response (or **`mplas_err_resp`** to send an error response).

**Function Prototype:** `ST_VOID u_mvl_namelist_ind (MVL_IND_PEND *ind_pend);`

**Parameters:**

`ind_pend` This is the same parameter that is passed to all user defined Indication functions. It contains a union of request and/or response parameters that is used for several different services. The request parameters are set by MVL before calling this function. The response parameters must be set by the user before calling the response function (i.e., **`mplas_namelist_resp`**). The parameters to be used for this service are as follows:

**Request parameters:**

| | |
|---|---|
| `ind_pend->u.namelist.req_info` | See **`NAMELIST_REQ_INFO`** for more information. |

**Response parameters:**

| | |
|---|---|
| `ind_pend->u.namelist.resp_info` | See **`NAMELIST_RESP_INFO`** for more information. |

**Return Value:** `ST_VOID`

## mplas_namelist_resp

**Usage:** This function encodes and sends the Response for a previously received GetNameList indication. The response parameters in the **MVL_IND_PEND** structure must be filled in before this function is called. The (**MVL_IND_PEND ***) argument passed to this function must be the same as the (**MVL_IND_PEND ***) argument passed to the user defined function **u_mvl_namelist_ind** when the indication was received.

**Function Prototype:** `ST_VOID mplas_namelist_resp (MVL_IND_PEND *ind_pend);`

**Parameters:**

ind_pend

This is the same parameter that is passed to all user defined Indication functions. It contains a union of request and/or response parameters that is used for several different services. The request parameters are set by MVL before calling this function. The response parameters must be set by the user before calling the response function (i.e., **mplas_namelist_resp**). The parameters to be used for this service are as follows:

**Request parameters:**

| | |
|---|---|
| ind_pend->u.namelist.req_info | See **NAMELIST_REQ_INFO** for more information. |

**Response parameters:**

| | |
|---|---|
| ind_pend->u.namelist.resp_info | See **NAMELIST_RESP_INFO** for more information. |

**Return Value:** ST_VOID

## mvlas_namelist_resp

**Usage:** This is a Virtual Machine response function for handling a previously received GetNameList indication. It completely processes the indication, fills in the response parameters in the **`MVL_IND_PEND`** structure, and then sends the response by calling **`mplas_namelist_resp`**. The (**`MVL_IND_PEND *`**) argument passed to this function must be the same as the (**`MVL_IND_PEND *`**) argument passed to the user defined function **`u_mvl_namelist_ind`** when the indication was received.

**Function Prototype:** `ST_VOID mvlas_namelist_resp (MVL_IND_PEND *ind_pend);`

**Parameters:**

`ind_pend` This is the same parameter that is passed to all user defined Indication functions. It contains a union of request and/or response parameters that is used for several different services. The request parameters are set by MVL before calling this function. The parameters to be used for this service are as follows:

**Request parameters:**

| | |
|---|---|
| `ind_pend->u.namelist.req_info` | See **`NAMELIST_REQ_INFO`** for more information. |

**Response parameters:**

| | |
|---|---|
| `ind_pend->u.namelist.resp_info` | See **`NAMELIST_RESP_INFO`** for more information. |

**Return Value:** `ST_VOID`

# GetCapabilityList Service

This service is used to request that a list (or part of a list) of capabilities that exist at the VMD.

## *GetCapabilityList Data Structures*

The following is the GetCapabilityList Indication Control Structure:

```
typedef struct
  {
  GETCL_REQ_INFO *req_info;
  GETCL_RESP_INFO *resp_info;
  } MVLAS_GETCL_CTRL;
```

### GETCL_REQ_INFO

This operation specific structure described below is used by the client in issuing a GetCapabilityList request. It is received by the server when a GetCapabilityList indication is received.

```
struct getcl_req_info
  {
  ST_BOOLEAN cont_after_pres;
  ST_CHAR *continue_after;
  };
typedef struct getcl_req_info GETCL_REQ_INFO;
```

**<u>Fields</u>**:

`cont_after_pres` **`SD_FALSE`**. Do not include the **`continue_after`** field in the PDU. Begin the capability list response from the beginning of the list. **`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.

### GETCL_RESP_INFO

This operation specific structure described below is used by the server in issuing a GetCapabilityList response. It is received by the client when a GetCapabilityList confirm 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;
```

**<u>Fields</u>**:

`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 *));
```

### *GetCapabilityList Functions*

## u_mvl_getcl_ind

**Usage:** This user function is called by MVL when a Get Capability List indication is received. It should build the response and call **`mplas_getcl_resp`** to send the response. See the file **server.c** for an example of this function.

**Function Prototype:** `ST_VOID u_mvl_getcl_ind (MVL_IND_PEND *ind_pend);`

**Parameters:**

`ind_pend` This is the same parameter that is passed to all user defined Indication functions. It contains a union of request and/or response parameters that is used for several different services. The request parameters are set by MVL before calling this function. The response parameters must be set by the user before calling the response function (i.e., **`mplas_getcl_resp`**). The parameters to be used for this service are as follows:

**Request parameters:**

| | |
|---|---|
| `ind_pend->u.getcl.req_info` | See **`GETCL_REQ_INFO`** for more information. |

**Response parameters:**

| | |
|---|---|
| `ind_pend->u.getcl.resp_info` | See **`GETCL_RESP_INFO`** for more information. |

**Return Value:** `ST_VOID`

## mplas_getcl_resp

**Usage:** This function encodes and sends the Response for a previously received GetCapabilityList indication. The response parameters in the **MVL_IND_PEND** structure must be filled in before this function is called. The (**MVL_IND_PEND ***) argument passed to this function must be the same as the (**MVL_IND_PEND ***) argument passed to the user defined function **u_mvl_getcl_ind** when the indication was received.

**Function Prototype:** `ST_VOID mplas_getcl_resp (MVL_IND_PEND *ind_pend);`

**Parameters:**

`ind_pend` This is the same parameter that is passed to all user defined Indication functions. It contains a union of request and/or response parameters that is used for several different services. The request parameters are set by MVL before calling this function. The response parameters must be set by the user before calling the response function (i.e., **mplas_getcl_resp**). The parameters to be used for this service are as follows:

**Request parameters:**

| | |
|---|---|
| `ind_pend->u.getcl.req_info` | See **GETCL_REQ_INFO** for more information. |

**Response parameters:**

| | |
|---|---|
| `ind_pend->u.getcl.resp_info` | See **GETCL_RESP_INFO** for more information. |

**Return Value:** `ST_VOID`

# Variable Access Overview

MVL provides a set of flexible mechanisms for supporting the MMS Variable Access services effectively in the application, including:

- Configured or manufactured Variable Associations or a combination of both
- Variable Association pre/post processing functions for Read and Write services
- Alternate Access Support

## Variable Association

MVL uses a construct called a Variable Association, which is used to map local variables and processes to MMS Named variables. MMS-EASE *Lite* has an easy to use mechanism for associating MMS Named Variables to system values and/or memory locations. The data structure that implements this is **MVL_VAR_ASSOC**. Each MMS-EASE *Lite* variable can have user-defined pre/post functions for the various MMS variable access services and can be dynamically desired. MMS-EASE *Lite* easily supports arbitrary data types of any complexity.

```
typedef struct mvl_var_assoc
  {
  ST_CHAR *name;                /* variable name                  */
  ST_VOID *data;                /* pointer to local data          */
  ST_INT type_id;               /* type of variable               */
  ST_UCHAR flags;               /* MVL_VAR_FLAG_UCA, etc.         */
  MVL_VAR_PROC *proc;         /* User defined pre/post processing */
  ST_VOID *user_info;   /* MVL user can use this for 'whatever'   */
  ST_VOID *usr_ind_ctrl;
#if defined(MVL_UCA) || defined(USE_MANUFACTURED_OBJS)
  struct mvl_var_assoc *va_to_free; /* Used in NVL processing     */
#endif
#if defined(MVL_UCA)
  struct mvl_var_assoc *base_va; /* VA from which this was derived */
  ST_INT offset_from_base;     /* Used only for static data buffer */
  ST_RTREF ref;
  MVL_ARR_CTRL arrCtrl;
  ST_VOID *mvl_internal;     /* ptr to info used internally by MVL */
  ST_BOOLEAN use_static_data; /* “data” in this struct points to   */
                              /* permanent data.                   */
#endif
#ifdef MVL_INFO_RPT_CLIENT
  ST_RET result;
#endif
  } MVL_VAR_ASSOC;
```

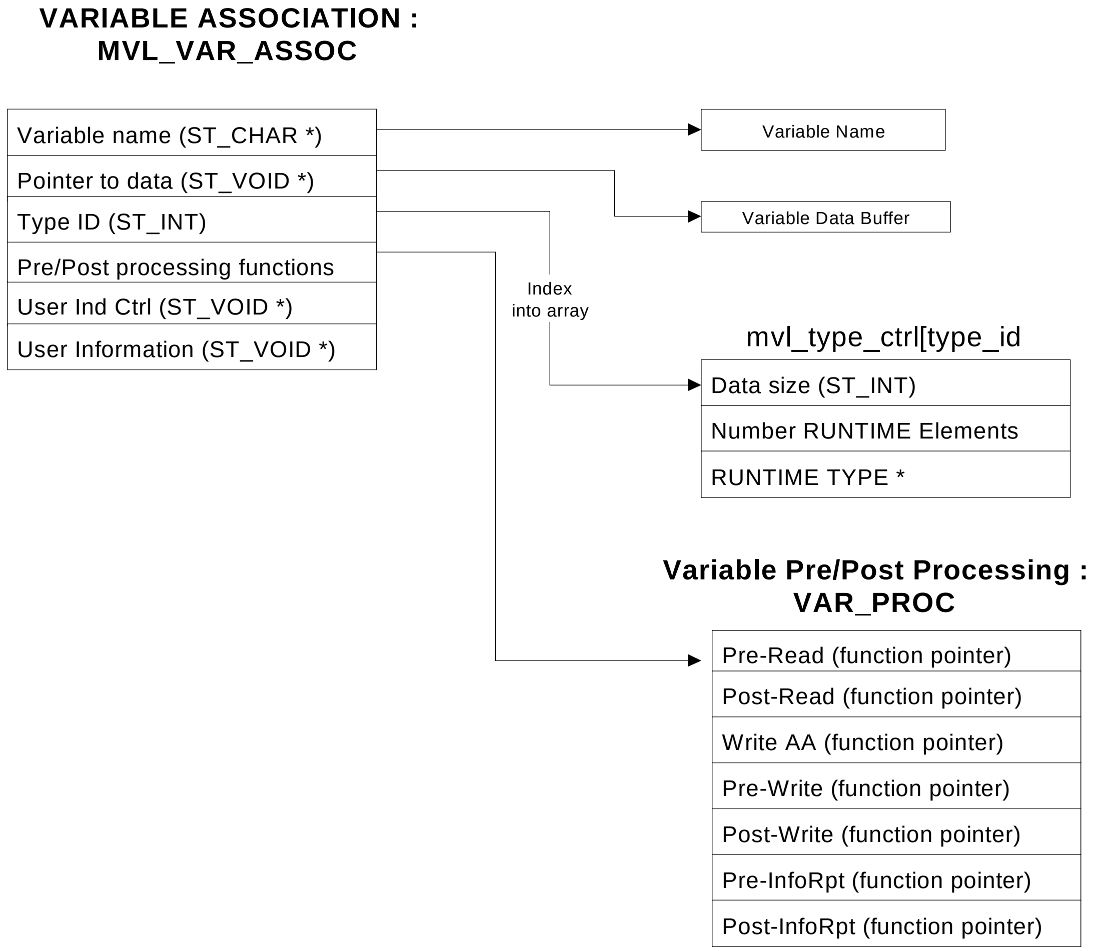

VARIABLE ASSOCIATION :
MVL_VAR_ASSOC
Variable name (ST_CHAR *)
Pointer to data (ST_VOID *)
Type ID (ST_INT)
Pre/Post processing functions
User Ind Ctrl (ST_VOID *)
User Information (ST_VOID *)
Variable Name
Variable Data Buffer
Index
into array
mvl_type_ctrl[type_id
Data size (ST_INT)
Number RUNTIME Elements
RUNTIME TYPE *
Variable Pre/Post Processing :
VAR_PROC
Pre-Read (function pointer)
Post-Read (function pointer)
Write AA (function pointer)
Pre-Write (function pointer)
Post-Write (function pointer)
Pre-InfoRpt (function pointer)
Post-InfoRpt (function pointer)

Figure 6: MVL Variable Association Data Structure

## *Configuring Named Variables*

MVL allows the developer to configure MMS Named Variables using **MMS Object Foundry**, tool for creating MMS server objects, which most applications can and should make use of. The configured variable approach is the easiest to implement. The developer simply supplies the MMS variable names and linkages to application variables using the MMS Object Foundry **Object Definition File**. MVL will then manage all aspects of the variables automatically. The application can make use of pre/post processing functions that may be attached to any variable as well as the MVL asynchronous **Read** response capability. For instance, to expose the local variable "Temperature" as a MMS variable, the following line can be added to the **ODF** file:

```
":VD", "Temperature", "I16_TYPE",   "&Temperature"
```

This will result in MVL exposing the local variable "Temperature" as a MMS variable "Temperature", with all further application programming optional. Please refer to page 271 for more information on configuring variables.

## *Configuring Named Variable Lists*

As with named variables, MVL allows the developer to configure MMS Named Variable Lists (NVLs) using **MMS Object Foundry**. Again, most applications can and should make use of this facility. The configured NVL approach is the easiest to implement; the developer simply declares a MMS NVL and provides a list of configured Named Variables for the list using the **MMS Object Foundry Object Definition File**. MVL will then manage all aspects of this NVL automatically. Note that to the application, variable access using either **list of variables** or **named variable list** access specifications are handled at the **Variable Association** level.

For example, to create a NVL called "TwoVars" with the MMS variables "Temperature" and "Pressure", the following line can be added to the **ODF** file:

```
":L", "TwoVars", "Temperature", "Pressure",  ":S"
```

Please refer to page 273 for more information on configuring Named Variable Lists.

## *Manufactured Variables*

MVL provides the developer with the option of **manufacturing** variables instead of configuring them. This means that the application will have MMS visible variables that do not have static Variable Associations (VA). To enable this feature, the MVL library and the application must be compiled with **`USE_MANUFACTURED_OBJS`** defined. If this is defined, then the user function **`u_mvl_get_va_aa`** is called to resolve the MMS variable name to a **manufactured** Variable Association. This Variable Association must be committed until the response is sent and the function **`u_mvl_free_va`** is invoked to allow the application to free the VA and any associated resources. Note that the parameter **`alt_access_done_out`** can be set **`SD_TRUE`** if the function has resolved the alternate access when generating the Variable Association.

When using manufactured variables, the application becomes responsible for handling the MMS GetNameList (GNL) service for Named Variables and must provide the user function **`u_gnl_ind_vars`**.

## *Manufactured Named Variable Lists*

As with Manufactured Variables, MVL allows the application to manufacture Named Variable Lists. The user functions **`u_mvl_get_nvl`** on page 67 and **`u_mvl_free_nvl`** on page 68 are called to create and destroy Named Variable Lists. Note that a manufactured NVL must contain valid references to its associated Variable Associations, which may or may not be manufactured as well. Like Manufactured Variables, the application should install a MMS GetNameList handler for Named Variable Lists via the function pointer **`u_gnl_ind_nvls`**.

## *Alternate Access*

An application can make use of MVL support for the MMS Alternate Access feature in two ways:

First, MVL can handle the entire Alternate Access resolution transparently to the user. Complete the following steps to include this support:

1. Compile **mmsdataa.c** and link it into your application. This will replace the stub functions in the MMS library.
2. Edit **mms_tdef.c**. Find or add a definition of the following array that corresponds to the target platform. Note that this is the same text as in the MMS Object Foundry **align.cfg** file.

   ```
   ST_INT m_def_data_algn_tbl [NUM_ALGN_TYPES]
   ```

3. Make sure **mms_tdef.c**, **mms_alta.c**, and **mms_rtaa.c** are included in the MMS library.

When this is done, no further action is required from the user application. The primary downside to this is the necessity of linking **mmsdataa.c**, which is rather resource intensive. In addition, this method does not work especially well with manufactured variables.

With the second method, the user can handle the alternate access and deal with it outside of MVL. That is, it can be handled in the pre-read, pre-write, or manufactured variable handlers as appropriate. This approach is preferable for simple types of alternate access and when the application uses the manufactured variable mechanism.

## Read Service

MVL has flexible and easy to use support for the MMS Read service. The general flow and user options are shown on the flowcharts below.

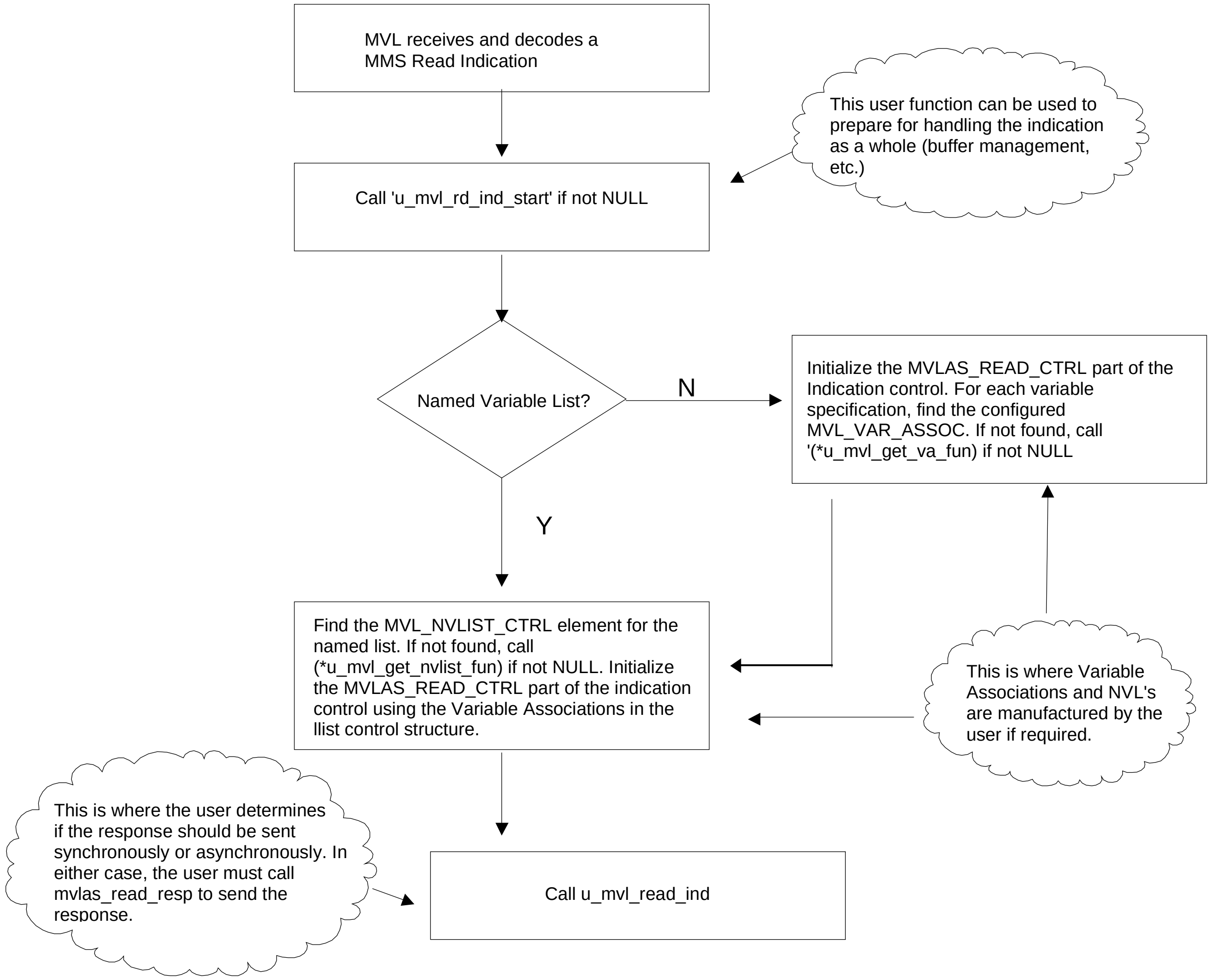

MVL receives and decodes a MMS Read Indication
Call 'u_mvl_rd_ind_start' if not NULL
This user function can be used to prepare for handling the indication as a whole (buffer management, etc.)
Named Variable List?
N
Initialize the MVLAS_READ_CTRL part of the Indication control. For each variable specification, find the configured MVL_VAR_ASSOC. If not found, call '(*u_mvl_get_va_fun) if not NULL
Y
Find the MVL_NVLIST_CTRL element for the named list. If not found, call (*u_mvl_get_nvlist_fun) if not NULL. Initialize the MVLAS_READ_CTRL part of the indication control using the Variable Associations in the llist control structure.
This is where Variable Associations and NVL's are manufactured by the user if required.
This is where the user determines if the response should be sent synchronously or asynchronously. In either case, the user must call mvlas_read_resp to send the response.
Call u_mvl_read_ind

Figure 7: MVL Read Indication Processing

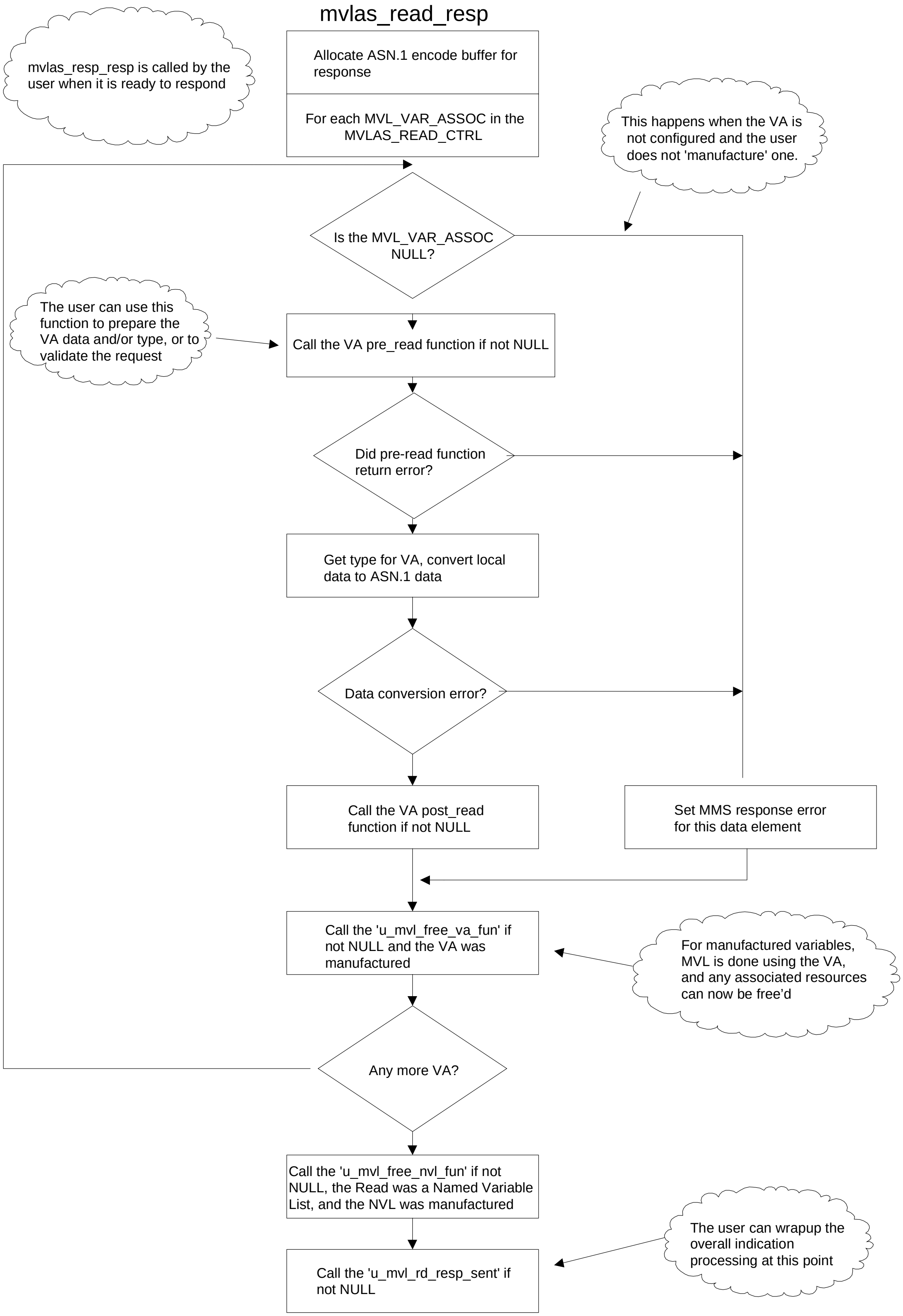

mvlas_read_resp
mvlas_resp_resp is called by the user when it is ready to respond
Allocate ASN.1 encode buffer for response
For each MVL_VAR_ASSOC in the MVLAS_READ_CTRL
This happens when the VA is not configured and the user does not 'manufacture' one.
Is the MVL_VAR_ASSOC NULL?
The user can use this function to prepare the VA data and/or type, or to validate the request
Call the VA pre_read function if not NULL
Did pre-read function return error?
Get type for VA, convert local data to ASN.1 data
Data conversion error?
Call the VA post_read function if not NULL
Set MMS response error for this data element
Call the 'u_mvl_free_va_fun' if not NULL and the VA was manufactured
For manufactured variables, MVL is done using the VA, and any associated resources can now be free'd
Any more VA?
Call the 'u_mvl_free_nvl_fun' if not NULL, the Read was a Named Variable List, and the NVL was manufactured
The user can wrapup the overall indication processing at this point
Call the 'u_mvl_rd_resp_sent' if not NULL

Figure 8: MVL Read Response Processing

## *Read Data Structures*

The following are the Read Indication Control Structures.

This structure represents one Variable Association being read. The user can set the **`acc_rslt_tag`** to **`ACC_RSLT_FAILURE`** if the read does not succeed.

### MVLAS_RD_VA_CTRL

```
typedef struct mvlas_rd_va_ctrl
  {
  MVL_VAR_ASSOC *va;
  MVL_SCOPE va_scope;         /* Variable scope: VMD, Domain, or AA */
  ST_INT16 acc_rslt_tag;      /* ACC_RSLT_SUCCESS or ACC_RSLT_FAILURE */
  ST_INT16 failure;           /* DataAccessError code for failure     */
  ST_BOOLEAN alt_access_pres;
  ALT_ACCESS alt_acc;

#if defined(MVL_UCA)
  ST_INT numPrimData;         /* Total primitive elements for var     */
  ST_INT numPrimDataDone;     /* Number complete                      */
#endif

  ST_VOID *usr;               /* For user to use as seen fit          */
  } MVLAS_RD_VA_CTRL;
```

### MVLAS_READ_CTRL

```
typedef struct mvlas_read_ctrl
  {
  ST_INT16 var_acc_tag;   /* VAR_ACC_NAMEDLIST or VAR_ACC_VARLIST */
  ST_INT numVar;          /* Variables being read                 */
  MVLAS_RD_VA_CTRL *vaCtrlTbl;

  ST_BOOLEAN usrNvl;          /* MVL internal use                 */
  MVL_NVLIST_CTRL *nvList;
  } MVLAS_READ_CTRL;
```

## *Read Functions*

## u_mvl_read_ind

**Usage:** This is a user-defined function called when a Read indication is received. The user must call **`mvlas_read_resp`** to automatically process the indication and send the response or call **`mplas_err_resp`** to send an error response.

**Function Prototype:** `ST_VOID u_mvl_read_ind (MVL_IND_PEND *ind_pend);`

**Parameters:**

`ind_pend` This is the same parameter that is passed to all user defined Indication functions. It contains a union of request and/or response parameters that is used for several different services. The request parameters are set by MVL before calling this function. The response parameters must be set by the user before calling the response function. The parameters to be used for this service are as follows:

**Request and response parameters:**

| | |
|---|---|
| `ind_pend->u.rd` | See **`MVLAS_READ_CTRL`** for more information. |

**Return Value:** `ST_VOID`

## mvlas_read_resp

**Usage:** This is a Virtual Machine response function for handling a previously received Read indication. It completely processes the indication, fills in the response parameters in the **MVL_IND_PEND** structure, and then sends the response. This function is usually called synchronously from the **u_mvl_read_ind** function, but it may be called asynchronously whenever the service is completed. The (**MVL_IND_PEND ***) argument passed to this function must be the same as the (**MVL_IND_PEND ***) argument passed to the user defined function **u_mvl_read_ind** when the indication was received.

**Function Prototype:** `ST_RET mvlas_read_resp (MVL_IND_PEND *ind_pend);`

**Parameters:**

`ind_pend` This is the same parameter that is passed to all user defined Indication functions. It contains a union of request and/or response parameters that is used for several different services. The request parameters are set by MVL before calling this function. The parameters to be used for this service are as follows:

**Request and response parameters:**

| | |
|---|---|
| `ind_pend->u.rd` | See **MVLAS_READ_CTRL** for more information. |

**Return Value:** `ST_RET` `SD_SUCCESS` If OK, or a non-zero error code.

## *Read Pre/Post Processing Functions*

MVL provides a facility for user defined pre/post processing functions for all server variables. Each variable can have independent pre/post processing functions associated with it. These are referenced via a structure of type **`MVL_VAR_PROC`**.

### pre_read

This function is called just before MVL encodes the ASN.1 data for the read response. MVL uses the information in the Variable Association (VA) in encoding the ASN.1 data.

The task for this function is to ensure that the VA is valid so that MVL can send the correct data in response to a read indication. This function is passed the VA and any alternate access information and can take whatever steps are required to resolve the VA data, type, and alternate access mode to be used. Some typical steps taken by this function can include:

- Return a different VA to be used.
- Change the data buffer (i.e., **`va->data = newDataPtr`**).
- Change the data in the data buffer.
- Change the type (i.e., **`va->type_id - newTypeId`**).
- Change the Alternate Access processing mode.

The **`pre_read`** function must return **`SD_SUCCESS`** (if the VA is ready to be used by MVL) or **`SD_FAILURE`** (if the VA is not ready to be used by MVL).

### post_read

This function is called after MVL has encoded the ASN.1 data for a read response. It may be used for application specific purposes, such as freeing resources used during the read process. This function is passed the VA and any alternate access information.

# Write Service

MVL has flexible and easy to use support for the MMS Write service. The general flow and user options are shown on the flowcharts below.

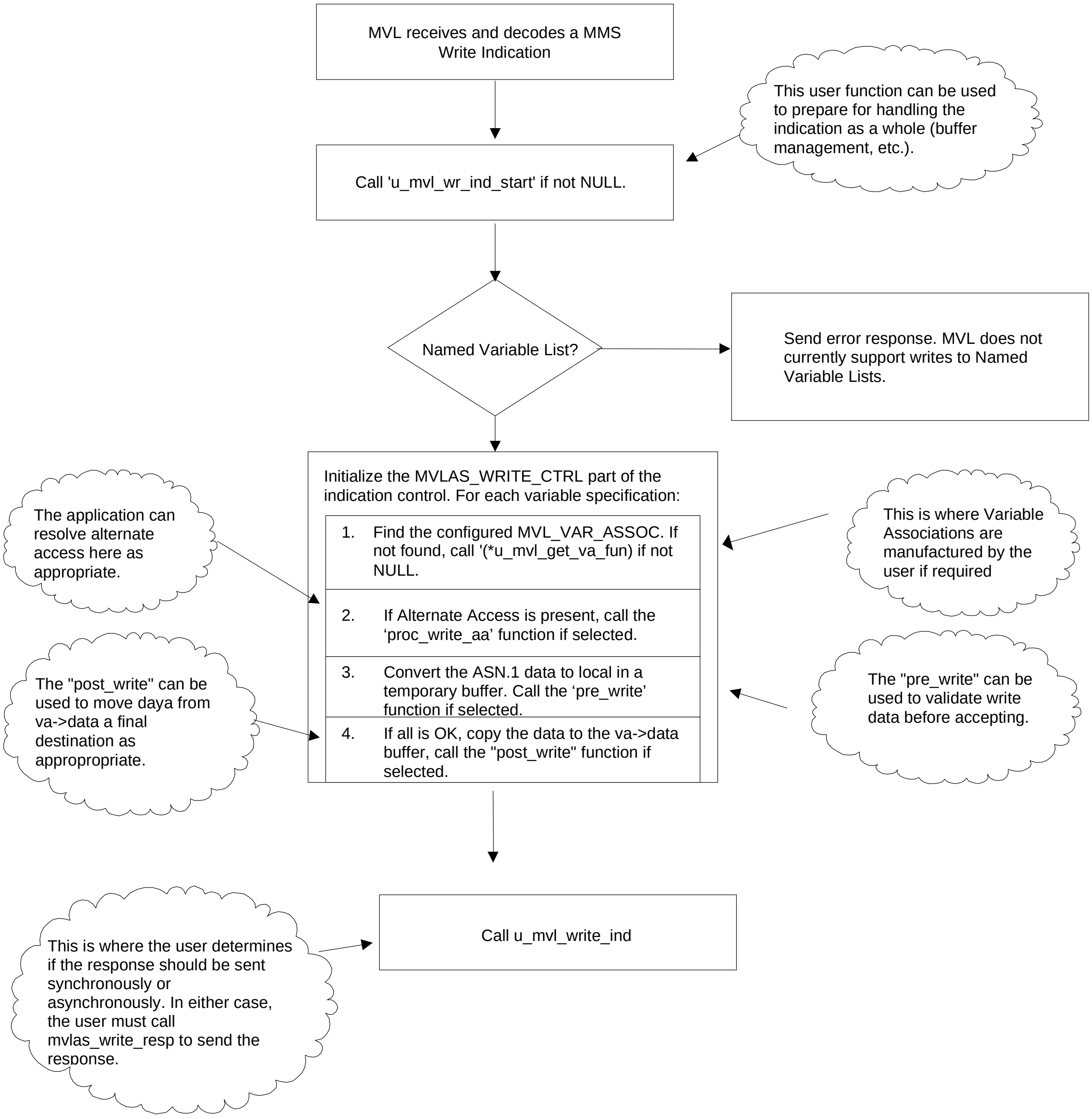

MVL receives and decodes a MMS Write Indication
Call 'u_mvl_wr_ind_start' if not NULL.
This user function can be used to prepare for handling the indication as a whole (buffer management, etc.).
Named Variable List?
Send error response. MVL does not currently support writes to Named Variable Lists.
Initialize the MVLAS_WRITE_CTRL part of the indication control. For each variable specification:
1. Find the configured MVL_VAR_ASSOC. If not found, call '(*u_mvl_get_va_fun) if not NULL.
2. If Alternate Access is present, call the 'proc_write_aa' function if selected.
3. Convert the ASN.1 data to local in a temporary buffer. Call the 'pre_write' function if selected.
4. If all is OK, copy the data to the va->data buffer, call the "post_write" function if selected.
The application can resolve alternate access here as appropriate.
This is where Variable Associations are manufactured by the user if required
The "post_write" can be used to move daya from va->data a final destination as appropropriate.
The "pre_write" can be used to validate write data before accepting.
Call u_mvl_write_ind
This is where the user determines if the response should be sent synchronously or asynchronously. In either case, the user must call mvlas_write_resp to send the response.

Figure 9: MVL Write Indication Processing

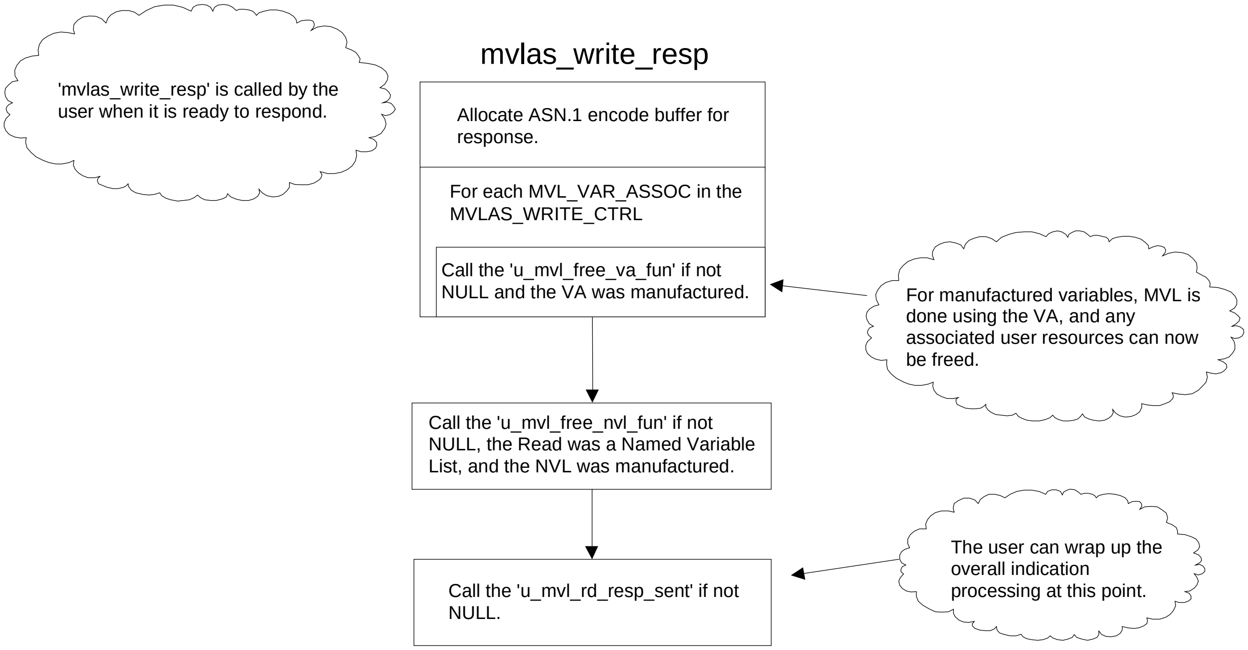

'mvlas_write_resp' is called by the user when it is ready to respond.
mvlas_write_resp
Allocate ASN.1 encode buffer for response.
For each MVL_VAR_ASSOC in the MVLAS_WRITE_CTRL
Call the 'u_mvl_free_va_fun' if not NULL and the VA was manufactured.
For manufactured variables, MVL is done using the VA, and any associated user resources can now be freed.
Call the 'u_mvl_free_nvl_fun' if not NULL, the Read was a Named Variable List, and the NVL was manufactured.
Call the 'u_mvl_rd_resp_sent' if not NULL.
The user can wrap up the overall indication processing at this point.

Figure 10: MVL Write Response Processing

## Write Data Structures

The following are the Write Indication Control Structures:

This structure represents one Variable Association being written. The user can set the **`resp_tag`** to **`WR_RSLT_FAILURE`** if the write does not succeed.

### MVLAS_WR_VA_CTRL

```
typedef struct mvlas_wr_va_ctrl
  {
  MVL_VAR_ASSOC *va;
  MVL_SCOPE va_scope;         /* Variable scope: VMD, Domain, or AA */
  ST_INT16 resp_tag;          /* WR_RSLT_FAILURE or WR_RSLT_SUCCESS */
  ST_INT16 failure;           /* DataAccessError code for failure   */
  ST_BOOLEAN alt_access_pres;
  ALT_ACCESS alc_acc;

#if defined(MVL_UCA)
  ST_INT numPrimData;         /* Total primitive elements for var */
  ST_INT numPrimDataDone;     /* Number complete                  */
#endif

  ST_VOID *usr;               /* For user to use as she sees fit  */
  } MVLAS_WR_VA_CTRL;
```

### MVLAS_WRITE_CTRL

```
typedef struct mvlas_write_ctrl
  {
  ST_INT numVar;        /* Variables being written                */
  MVLAS_WR_VA_CTRL *vaCtrlTbl;
  } MVLAS_WRITE_CTRL;
```

## *Write Functions*

### u_mvl_write_ind

**Usage:** This is a user defined function called when a Write indication is received. The user must call **`mvlas_write_resp`** to automatically process the indication and send the response or call **`mplas_err_resp`** to send an error response.

**Function Prototype:** `ST_VOID u_mvl_write_ind (MVL_IND_PEND *ind_pend);`

**Parameters:**

`ind_pend` This is the same parameter that is passed to all user defined Indication functions. It contains a union of request and/or response parameters that is used for several different services. The request parameters are set by MVL before calling this function. The response parameters must be set by the user before calling the response function. The parameters to be used for this service are as follows:

**Request and response parameters:**

| | |
|---|---|
| `ind_pend->u.wr` | See **`MVLAS_WRITE_CTRL`** for more information. |

**Return Value:** `ST_VOID`

## mvlas_write_resp

**Usage:** This is a Virtual Machine response function for handling a previously received Write indication. It completely processes the indication, fills in the response parameters in the **`MVL_IND_PEND`** structure, and then sends the response. This function is usually called synchronously from the **`u_mvl_write_ind`** function, but it may be called asynchronously whenever the service is completed. The (**`MVL_IND_PEND *`**) argument passed to this function must be the same as the (**`MVL_IND_PEND *`**) argument passed to the user defined function **`u_mvl_write_ind`** when the indication was received.

**Function Prototype:** `ST_VOID mvlas_write_resp (MVL_IND_PEND *ind_pend);`

**Parameters:**

`ind_pend` This is the same parameter that is passed to all user defined Indication functions. It contains a union of request and/or response parameters that is used for several different services. The request parameters are set by MVL before calling this function. The parameters to be used for this service are as follows:

**Request and response parameters:**

| | |
|---|---|
| `ind_pend->u.wr` | See **`MVLAS_WRITE_CTRL`** for more information. |

**Return Value:** `ST_VOID`

## *Write Pre/Post Processing Functions*

MVL provides the hooks for pre/post processing functions for all server variables. Each variable can have pre/post processing functions associated with it. These are referenced using a structure of type **`MVL_VAR_PROC`**.

### proc_write_aa

This function is called when a MMS Write indication has been received and alternate access is present for the Variable Association. MVL calls this function before converting the ASN.1 write data to local in a temporary buffer.

This function is passed the VA and the alternate access information, and can take whatever steps are required to prepare for the data conversion process. Some typical steps take by this function can include:

- Return a different VA to be used.
- Change the type.
- Change the Alternate Access processing mode.

The **`proc_write_aa`** function must return **`SD_SUCCESS`** (if the VA is ready to be used by MVL) or **`SD_FAILURE`** if not (if the VA is not ready to be used by MVL).

### pre_write

This function is called after MVL has decoded the received ASN.1 data into a temporary buffer. The task for this function is to check that the data is acceptable and that the VA is valid so that MVL can copy the data from the temporary buffer into the VA data buffer.

This function is passed the VA, the data, and any alternate access information. Some typical steps take by this function can include:

- Verify that the data is acceptable.
- Change the data buffer.

The **`pre_write`** function must return **`SD_SUCCESS`** (if the VA is ready to be used by MVL) or **`SD_FAILURE`** (if the VA is not ready to be used by MVL).

### post_write

This function is called after MVL has moved the decoded data into the VA data buffer. It may be used for application specific purposes, such as moving the data into a final destination. This function is passed the VA and any alternate access information. An example of how these hooks could be used is as follows:

Assume the server has a MMS server variable called **setPoint**. Whenever the server variable “setPoint” is written by a remote client, we want to check the value to determine whether it is valid; and when the local variable value has been changed, we want to call a routine that will take action on the new setPoint value. To do this, we can use the **`pre_write`** function for the validity check (return **`SD_SUCCESS`** if OK, **`SD_FAILURE`** if not) and the **`post_write`** function to take action on the new value.

# Information Report 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.

## *Information Report Functions*

### mvl_info_variables

**Usage:** This function is used to send a MMS Information Report. It takes a `MVL_NVLIST_CTRL` as input, and sends the values over the selected network connection as either List Of Variables (**`listOfVariables == SD_TRUE`**) or a Named Variable List (**`listOfVariables == SD_FALSE`**).

**Function Prototype:**

```
ST_RET mvl_info_variables (MVL_NET_INFO *net_info,
                           MVL_NVLIST_CTRL *nvl,
                           ST_BOOLEAN listOfVariables);
```

**Parameters:**

| | |
|---|---|
| `net_info` | This parameter selects the connection on which the MMS transaction is to take place. |
| `nvl` | This is the address of a Named Variable List object that contains the data to send in the InformationReport. It may be sent as a Named Variable List or as a List of Variables depending on the value of the *listOfVariables* argument. Note that in either case the MMS data will be the same. The structure **`MVL_NVLIST_CTRL`** is defined in **mvl_defs.h**. |
| `listOfVariables` | This parameter is used to select the form of the MMS Variable Specification to be sent. The value **`SD_TRUE`** will result in a MMS List Of Variables, **`SD_FALSE`** will result in a MMS Named Variable List. |

**Return Value:** `ST_RET` `SD_SUCCESS` If OK, or a non-zero error code.

# 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.

## *GetVariableAccessAttributes Data Structures*

The following is the GetvariableAccessAttributes Indication Control Structure:

```
typedef struct
  {
  GETVAR_REQ_INFO *req_info;
  GETVAR_RESP_INFO *resp_info;
  } MVLAS_GETVAR_CTRL;
```

### GETVAR_REQ_INFO

The operation-specific data structure is used by the Client in issuing a GetVariableAccessAttributes request. It is received by the Server when a GetVariableAccessAttributes indication 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`**.

### GETVAR_RESP_INFO

The operation-specific data structure described on the next page is used by the Server in issuing a GetVariableAccessAttributes response. It is received by the Client when a GetVariableAccessAttributes confirm 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.<br>**SD_TRUE**. The variable definition is deletable using a MMS service request. |
| address_pres | **SD_FALSE**. Do not include **address** in the PDU.<br>**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. |

### *GetVariableAccessAttributes Functions*

## u_mvl_getvar_ind

**Usage:** This is a user defined function called when a GetVariableAccessAttributes indication is received. The user must examine the request parameters contained in the **MVL_IND_PEND** structure, do whatever is necessary to process the request, and then call **mplas_getvar_resp** or **mvlas_getvar_resp** to send the response (or **mplas_err_resp** to send an error response).

**Function Prototype:** `ST_VOID u_mvl_getvar_ind (MVL_IND_PEND *ind_pend);`

**Parameters:**

ind_pend

This is the same parameter that is passed to all user defined Indication functions. It contains a union of request and/or response parameters that is used for several different services. The request parameters are set by MVL before calling this function. The response parameters must be set by the user before calling the response function (i.e., **mplas_getvar_resp**). The parameters to be used for this service are as follows:

**Request parameters:**

| | |
|---|---|
| ind_pend->u.getvar.req_info | See **GETVAR_REQ_INFO** for more information. |

**Response parameters:**

| | |
|---|---|
| ind_pend->u.getvar.resp_info | See **GETVAR_RESP_INFO** for more information. |

## mplas_getvar_resp

**Usage:** This function encodes and sends the Response for a previously received GetVariableAccessAttributes indication. The Response parameters in the **MVL_IND_PEND** structure must be filled in before this function is called. The (**MVL_IND_PEND ***) argument passed to this function must be the same as the (**MVL_IND_PEND ***) argument passed to the user defined function **u_mvl_getvar_ind** when the indication was received.

**Function Prototype:** `ST_VOID mplas_getvar_resp (MVL_IND_PEND *ind_pend);`

**Parameters:**

`ind_pend` This is the same parameter that is passed to all user defined Indication functions. It contains a union of request and/or response parameters that is used for several different services. The request parameters are set by MVL before calling this function. The response parameters must be set by the user before calling the response function (i.e., **mplas_getvar_resp**). The parameters to be used for this service are as follows:

**Request parameters:**

| | |
|---|---|
| `ind_pend->u.getvar.req_info` | See **GETVAR_REQ_INFO** for more information. |

**Response parameters:**

| | |
|---|---|
| `ind_pend->u.getvar.resp_info` | See **GETVAR_RESP_INFO** for more information. |

**Return Value:** `ST_VOID`

## mvlas_getvar_resp

**Usage:** This is a Virtual Machine response function for handling a previously received GetVariableAccessAttributes indication. It completely processes the indication, fills in the response parameters in the **`MVL_IND_PEND`** structure, and then sends the response by calling **`mplas_getvar_resp`**. The (**`MVL_IND_PEND *`**) argument passed to this function must be the same as the (**`MVL_IND_PEND *`**) argument passed to the user defined function **`u_mvl_getvar_ind`** when the indication was received.

**Function Prototype:** `ST_VOID mvlas_getvar_resp (MVL_IND_PEND *ind_pend);`

**Parameters:**

`ind_pend` This is the same parameter that is passed to all user defined Indication functions. It contains a union of request and/or response parameters that is used for several different services. The request parameters are set by MVL before calling this function. The parameters to be used for this service are as follows:

**Request parameters:**

| | |
|---|---|
| `ind_pend->u.getvar.req_info` | See **`GETVAR_REQ_INFO`** for more information. |

**Response parameters:**

| | |
|---|---|
| `ind_pend->u.getvar.resp_info` | See **`GETVAR_RESP_INFO`** for more information. |

**Return Value:** `ST_VOID`

# 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.

## *DefineNamedVariableList Data Structures*

The following is the DefinedNamedVariableList Indication Control Structure:

```
typedef struct
  {
  DEFVLIST_REQ_INFO *req_info;
  } MVLAS_DEFVLIST_CTRL;
```

### DEFVLIST_REQ_INFO

The operation-specific data structure described below is used by the Client in issuing a DefineNamedVariableList request. It is received by the Server when a DefineNamedVariableList indication 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.

***Note:*** *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))));
```

### *DefineNamedVariableList Functions*

## u_mvl_defvlist_ind

**Usage:** This is a user defined function called when a DefineNamedVariableList indication is received. The user must examine the request parameters contained in the **`MVL_IND_PEND`** structure, do whatever is necessary to process the request, and then call **`mplas_defvlist_resp`** or **`mvlas_defvlist_resp`** to send the response (or **`mplas_err_resp`** to send an error response).

**Function Prototype:** `ST_VOID u_mvl_defvlist_ind (MVL_IND_PEND *ind_pend);`

**Parameters:**

`ind_pend` This is the same parameter that is passed to all user defined Indication functions. It contains a union of request and/or response parameters that is used for several different services. The request parameters are set by MVL before calling this function. The parameters to be used for this service are as follows:

**Request parameters:**

| | |
|---|---|
| `ind_pend->u.defvlist.req_info` | See **`DEFVLIST_REQ_INFO`** for more information. |

**Response parameters:**

NONE

**Return Value:** `ST_VOID`

## mplas_defvlist_resp

**Usage:** This function encodes and sends the Response for a previously received DefineNamedVariableList indication. The (**`MVL_IND_PEND *`**) argument passed to this function must be the same as the (**`MVL_IND_PEND *`**) argument passed to the user defined function **`u_mvl_defvlist_ind`** when the indication was received.

**Function Prototype:** `ST_VOID mplas_defvlist_resp (MVL_IND_PEND *ind_pend);`

**Parameters:**

`ind_pend` This is the same parameter that is passed to all user defined Indication functions. It contains a union of request and/or response parameters that is used for several different services. The request parameters are set by MVL before calling this function. The parameters to be used for this service are as follows:

**Request parameters:**

| | |
|---|---|
| `ind_pend->u.defvlist.req_info` | See **`DEFVLIST_REQ_INFO`** for more information. |

**Response parameters:**

NONE

**Return Value:** `ST_VOID`

## mvlas_defvlist_resp

**Usage:** This is a Virtual Machine response function for handling a previously received DefineNamedVariableList indication. It completely processes the indication, fills in the response parameters in the **MVL_IND_PEND** structure, and then sends the response by calling **mplas_defvlist_resp**. The (**MVL_IND_PEND ***) argument passed to this function must be the same as the (**MVL_IND_PEND ***) argument passed to the user defined function **u_mvl_defvlist_ind** when the indication was received.

**Function Prototype:** `ST_VOID mvlas_defvlist_resp (MVL_IND_PEND *ind_pend);`

**Parameters:**

`ind_pend` This is the same parameter that is passed to all user defined Indication functions. It contains a union of request and/or response parameters that is used for several different services. The parameters to be used for this service are as follows:

**Request parameters:**

| | |
|---|---|
| `ind_pend->u.defvlist.req_info` | See **DEFVLIST_REQ_INFO** for more information. |

**Response parameters:**

NONE

**Return Value:** `ST_VOID`

# 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.

## *DeleteNamedVariableList Data Structures*

The following is the DeleteNamedVariableList Indication Control Structure:

```
typedef struct
  {
  DELVLIST_REQ_INFO *req_info;
  DELVLIST_RESP_INFO *resp_info;
  } MVLAS_DELVLIST_CTRL;
```

### DELVLIST_REQ_INFO

The operation-specific data structure described below is used by the Client in issuing a DeleteNamedVariableList request. It is received by the Server when a DeleteNamedVariableList indication 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. |

***Note:*** *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))));
```

### DELVLIST_RESP_INFO

The operation-specific data structure described below is used by the Server in issuing a DeleteNamedVariableList response. It is received by the Client when a DeleteNamedVariableList confirm 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. |

## *DeleteNamedVariableList Functions*

### u_mvl_delvlist_ind

**Usage:** This is a user defined function called when a DeleteNamedVariableList indication is received. The user must examine the request parameters contained in the **`MVL_IND_PEND`** structure, do whatever is necessary to process the request, and then call **`mplas_delvlist_resp`** or **`mvlas_delvlist_resp`** to send the response (or **`mplas_err_resp`** to send an error response).

**Function Prototype:** `ST_VOID u_mvl_delvlist_ind (MVL_IND_PEND *ind_pend);`

**Parameters:**

`ind_pend` This is the same parameter that is passed to all user defined Indication functions. It contains a union of request and/or response parameters that is used for several different services. The request parameters are set by MVL before calling this function. The response parameters must be set by the user before calling the response function (i.e., **`mvlas_delvlist_resp`** or **`mplas_delvlist_resp`**). The parameters to be used for this service are as follows:

**Request parameters:**

| | |
|---|---|
| `ind_pend->u.delvlist.req_info` | See **`DELVLIST_REQ_INFO`** for more information. |

**Response parameters:**

| | |
|---|---|
| `ind_pend->u. delvlist.resp_info` | See **`DELVLIST_RESP_INFO`** for more information. |

**Return Value:** `ST_VOID`

## mplas_delvlist_resp

**Usage:** This function encodes and sends the Response for a previously received DeleteNamedVariableList indication. The Response parameters in the **MVL_IND_PEND** structure must be filled in before this function is called. The (**MVL_IND_PEND ***) argument passed to this function must be the same as the (**MVL_IND_PEND ***) argument passed to the user defined function **u_mvl_delvlist_ind** when the indication was received.

**Function Prototype:** `ST_VOID mplas_delvlist_resp (MVL_IND_PEND *ind_pend);`

**Parameters:**

`ind_pend` This is the same parameter that is passed to all user defined Indication functions. It contains a union of request and/or response parameters that is used for several different services. The request parameters are set by MVL before calling this function. The response parameters must be set by the user before calling the response function (i.e., **mvlas_delvlist_resp** or **mplas_delvlist_resp**). The parameters to be used for this service are as follows:

**Request parameters:**

| | |
|---|---|
| ind_pend->u.delvlist.req_info | See **DELVLIST_REQ_INFO** for more information. |

**Response parameters:**

| | |
|---|---|
| ind_pend->u. delvlist.resp_info | See **DELVLIST_RESP_INFO** for more information. |

**Return Value:** `ST_VOID`

## mvlas_delvlist_resp

**Usage:** This is a Virtual Machine response function for handling a previously received DeleteNamedVariableList indication. It completely processes the indication, fills in the response parameters in the **`MVL_IND_PEND`** structure, and then sends the response by calling **`mplas_delvlist_resp`**. The (**`MVL_IND_PEND *`**) argument passed to this function must be the same as the (**`MVL_IND_PEND *`**) argument passed to the user defined function **`u_mvl_delvlist_ind`** when the indication was received.

**Function Prototype:** `ST_VOID mvlas_delvlist_resp (MVL_IND_PEND *ind_pend);`

**Parameters:**

`ind_pend` This is the same parameter that is passed to all user defined Indication functions. It contains a union of request and/or response parameters that is used for several different services. The request parameters are set by MVL before calling this function. The response parameters must be set by the user before calling the response function (i.e., **`mvlas_delvlist_resp`** or **`mplas_delvlist_resp`**). The parameters to be used for this service are as follows:

**Request parameters:**

| | |
|---|---|
| `ind_pend->u.delvlist.req_info` | See **`DELVLIST_REQ_INFO`** for more information. |

**Response parameters:**

| | |
|---|---|
| `ind_pend->u. delvlist.resp_info` | See **`DELVLIST_RESP_INFO`** for more information. |

**Return Value:** `ST_VOID`

# GetNamedVariableListAttributes

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.

## *GetNamedVariableListAttributes Data Structures*

The following is the GetNamedVariableListAttributes Indication Control Structure:

```
typedef struct
  {
  GETVLIST_REQ_INFO *req_info;
  GETVLIST_RESP_INFO *resp_info;    /* allocate appropriate size  */
  } MVLAS_GETVLIST_CTRL;
```

### GETVLIST_REQ_INFO

The operation-specific data structure described below is used by the Client in issuing a GetNamedVariableList Attributes request. It is received by the Server when a GetNamedVariableListAttributes indication 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.

### GETVLIST_RESP_INFO

The operation-specific data structure described below is used by the Server in issuing a GetNamedVariableList Attributes response. It is received by the Client when a GetNamedVariableListAttributes confirm 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 NamedVariableList 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))));
```

### *GetNamedVariableListAttributes Functions*

## u_mvl_getvlist_ind

**Usage:** This is a user defined function called when a GetNamedVariableListAttributes indication is received. The user must examine the request parameters contained in the **`MVL_IND_PEND`** structure, do whatever is necessary to process the request, and then call **`mplas_getvlist_resp`** or **`mvlas_getvlist_resp`** to send the response (or **`mplas_err_resp`** to send an error response).

**Function Prototype:** `ST_VOID u_mvl_getvlist_ind (MVL_IND_PEND *ind_pend);`

**Parameters:**

`ind_pend` This is the same parameter that is passed to all user defined Indication functions. It contains a union of request and/or response parameters that is used for several different services. The request parameters are set by MVL before calling this function. The response parameters must be set by the user before calling the response function (i.e., **`mvlas_getvlist_resp`** or **`mplas_getvlist_resp`**). The parameters to be used for this service are as follows:

**Request parameters:**

| | |
|---|---|
| `ind_pend->u.getvlist.req_info` | See **`GETVLIST_REQ_INFO`** for more information. |

**Response parameters:**

| | |
|---|---|
| `ind_pend->u.getvlist.resp_info` | See **`GETVLIST_RESP_INFO`** for more information. |

**Return Value:** `ST_VOID`

## mplas_getvlist_resp

**Usage:** This function encodes and sends the Response for a previously received GetNamedVariableListAttributes indication. The Response parameters in the **`MVL_IND_PEND`** structure must be filled in before this function is called. The (**`MVL_IND_PEND *`**) argument passed to this function must be the same as the (**`MVL_IND_PEND *`**) argument passed to the user defined function **`u_mvl_getvlist_ind`** when the indication was received.

**Function Prototype:** `ST_VOID mplas_getvlist_resp (MVL_IND_PEND *ind_pend);`

**Parameters:**

`ind_pend` This is the same parameter that is passed to all user defined Indication functions. It contains a union of request and/or response parameters that is used for several different services. The request parameters are set by MVL before calling this function. The response parameters must be set by the user before calling the response function (i.e., **`mvlas_getvlist_resp`** or **`mplas_getvlist_resp`**). The parameters to be used for this service are as follows:

**Request parameters:**

| | |
|---|---|
| `ind_pend->u.getvlist.req_info` | See **`GETVLIST_REQ_INFO`** for more information. |

**Response parameters:**

| | |
|---|---|
| `ind_pend->u.getvlist.resp_info` | See **`GETVLIST_RESP_INFO`** for more information. |

**Return Value:** `ST_VOID`

## mvlas_getvlist_resp

**Usage:** This is a Virtual Machine response function for handling a previously received GetNamedVariableListAttributes indication. It completely processes the indication, fills in the response parameters in the **MVL_IND_PEND** structure, and then sends the response by calling **mplas_getvlist_resp**. The (**MVL_IND_PEND ***) argument passed to this function must be the same as the (**MVL_IND_PEND ***) argument passed to the user defined function **u_mvl_getvlist_ind** when the indication was received.

**Function Prototype:** `ST_VOID mvlas_getvlist_resp (MVL_IND_PEND *ind_pend);`

**Parameters:**

`ind_pend` This is the same parameter that is passed to all user defined Indication functions. It contains a union of request and/or response parameters that is used for several different services. The request parameters are set by MVL before calling this function. The response parameters must be set by the user before calling the response function (i.e., **mvlas_getvlist_resp** or **mplas_getvlist_resp**). The parameters to be used for this service are as follows:

**Request parameters:**

| | |
|---|---|
| `ind_pend->u.getvlist.req_info` | See **GETVLIST_REQ_INFO** for more information. |

**Response parameters:**

| | |
|---|---|
| `ind_pend->u.getvlist.resp_info` | See **GETVLIST_RESP_INFO** for more information. |

**Return Value:** `ST_VOID`

# GetDomainAttributes Service

This service is used to request that a Server return all the attributes associated with a specified domain.

## GetDomainAttributes Data Structures

The following is the GetDomainAttributes Indication Control Structure:

```
typedef struct
  {
  GETDOM_REQ_INFO *req_info;
  GETDOM_RESP_INFO *resp_info;
  } MVLAS_GETDOM_CTRL;
```

### GETDOM_REQ_INFO

The operation specific structure described below is used by the Client in issuing a GetDomainAttributes request. It is received by the Server when a GetDomainAttributes indication 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. |

### GETDOM_RESP_INFO

This operation specific data structure described below is used by the Server in issuing a GetDomainAttributes response. It is received by the Client when a GetDomainAttributes confirm 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 *))));
```

## *GetDomainAttributes Functions*

### u_mvl_getdom_ind

**Usage:** This is a user defined function called when a GetDomainAttributes indication is received. The user must examine the request parameters contained in the **`MVL_IND_PEND`** structure, do whatever is necessary to process the request, and then call **`mplas_getdom_resp`** or **`mvlas_getdom_resp`** to send the response (or **`mplas_err_resp`** to send an error response).

**Function Prototype:** `ST_VOID u_mvl_getdom_ind (MVL_IND_PEND *ind_pend);`

`ind_pend` This is the same parameter that is passed to all user defined Indication functions. It contains a union of request and/or response parameters that is used for several different services. The request parameters are set by MVL before calling this function. The response parameters must be set by the user before calling the response function (i.e., **`mplas_getdom_resp`**). The parameters to be used for this service are as follows:

**Request parameters:**

| | |
|---|---|
| `ind_pend->u.getdom.req_info` | See **`GETDOM_REQ_INFO`** for more information. |

**Response parameters:**

| | |
|---|---|
| `ind_pend->u.getdom.resp_info` | See **`GETDOM_RESP_INFO`** for more information. |

**Return Value:** `ST_VOID`

## mplas_getdom_resp

**Usage:** This function encodes and sends the Response for a previously received GetDomainAttributes indication. The Response parameters in the **`MVL_IND_PEND`** structure must be filled in before this function is called. The (**`MVL_IND_PEND*`**) argument passed to this function must be the same as the (**`MVL_IND_PEND*`**) argument passed to the user defined function **`u_mvl_getdom_ind`** when the indication was received.

---

**Function Prototype:** `ST_VOID mplas_getdom_resp (MVL_IND_PEND *ind_pend);`

---

**Parameters:**

`ind_pend` This is the same parameter that is passed to all user defined Indication functions. It contains a union of request and/or response parameters that is used for several different services. The request parameters are set by MVL before calling this function. The response parameters must be set by the user before calling the response function (i.e., **`mplas_getdom_resp`**). The parameters to be used for this service are as follows:

**Request parameters:**

| | |
|---|---|
| `ind_pend->u.getdom.req_info` | See **`GETDOM_REQ_INFO`** for more information. |

**Response parameters:**

| | |
|---|---|
| `ind_pend->u.getdom.resp_info` | See **`GETDOM_RESP_INFO`** for more information. |

---

**Return Value:** `ST_VOID`

## mvlas_getdom_resp

**Usage:** This is a Virtual Machine response function for handling a previously received GetDomainAttributes indication. It completely processes the indication, fills in the response parameters in the **`MVL_IND_PEND`** structure, and then sends the response by calling **`mplas_getdom_resp`**. The (**`MVL_IND_PEND *`**) argument passed to this function must be the same as the (**`MVL_IND_PEND *`**) argument passed to the user defined function **`u_mvl_getdom_ind`** when the indication was received.

**Function Prototype:** `ST_VOID mvlas_getdom_resp (MVL_IND_PEND *ind_pend);`

**Parameters:**

`ind_pend` This is the same parameter that is passed to all user defined Indication functions. It contains a union of request and/or response parameters that is used for several different services. The request parameters are set by MVL before calling this function. The response parameters must be set by the user before calling the response function (i.e., **mplas_getdom_resp**). The parameters to be used for this service are as follows:

**Request parameters:**

| | |
|---|---|
| `ind_pend->u.getdom.req_info` | See **`GETDOM_REQ_INFO`** for more information. |

**Response parameters:**

| | |
|---|---|
| `ind_pend->u.getdom.resp_info` | See **`GETDOM_RESP_INFO`** for more information. |

**Return Value:** `ST_VOID`

# 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.

## *InitializeJournal Data Structures*

The following is the IntitializeJournal Indication Control Structure:

```
typedef struct
  {
  JINIT_REQ_INFO *req_info;
  JINIT_RESP_INFO *resp_info;
  } MVLAS_JINIT_CTRL;
```

### JINIT_REQ_INFO

The operation-specific structure described below is used by the client in issuing an InitializeJournal request. It is received by the server when an InitializeJournal indication 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.

`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.

### JINIT_RESP_INFO

The operation-specific data structure described below is used by the server in issuing an InitializeJournal response. It is received by the client when an InitializeJournal confirm 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.

## *InitializeJournal Functions*

### u_mvl_jinit_ind

**Usage:** This is a user defined function called when a InitializeJournal indication is received. The user must examine the request parameters contained in the **MVL_IND_PEND** structure, do whatever is necessary to process the request, fill in the response parameters in the **MVL_IND_PEND** structure, and then call **mplas_jinit_resp** to send the response (or **mplas_err_resp** to send an error response).

**Function Prototype:** `ST_VOID u_mvl_jinit_ind (MVL_IND_PEND *ind_pend);`

**Parameters:**

`ind_pend` This is the same parameter that is passed to all user defined Indication functions. It contains a union of request and/or response parameters that is used for several different services. The request parameters are set by MVL before calling this function. The response parameters must be set by the user before calling the response function (i.e., **mplas_jinit_resp**). The parameters to be used for this service are as follows:

**Request parameters:**

| | |
|---|---|
| `ind_pend->u.jinit.req_info` | See **JINIT_REQ_INFO** for more information. |

**Response parameters:**

| | |
|---|---|
| `ind_pend->u.jinit.resp_info` | See **JINIT_RESP_INFO** for more information. |

**Return Value:** `ST_VOID`

## mplas_jinit_resp

**Usage:** This function encodes and sends the Response for a previously received InitializeJournal indication. The Response parameters in the **`MVL_IND_PEND`** structure must be filled in before this function is called. The (**`MVL_IND_PEND *`**) argument passed to this function must be the same as the (**`MVL_IND_PEND *`**) argument passed to the user defined function **`u_mvl_jinit_ind`** when the indication was received.

**Function Prototype:** `ST_VOID mplas_jinit_resp (MVL_IND_PEND *ind_pend);`

**Parameters:**

`ind_pend` This is the same parameter that is passed to all user defined Indication functions. It contains a union of request and/or response parameters that is used for several different services. The request parameters are set by MVL before calling this function. The response parameters must be set by the user before calling the response function (i.e., **`mplas_jinit_resp`**). The parameters to be used for this service are as follows:

**Request parameters:**

| | |
|---|---|
| `ind_pend->u.jinit.req_info` | See **`JINIT_REQ_INFO`** for more information. |

**Response parameters:**

| | |
|---|---|
| `ind_pend->u.jinit.resp_info` | See **`JINIT_RESP_INFO`** for more information. |

**Return Value:** `ST_VOID`

# 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.

## *ReadJournal Data Structures*

The following is the ReadJournal Indication Control Structure:

```
typedef struct
  {
  JREAD_REQ_INFO *req_info;
  JREAD_RESP_INFO *resp_info;
/* Variable size. User or mvlas_* must alloc.   */
  } MVLAS_JREAD_CTRL;
```

### JREAD_REQ_INFO

The operation-specific structure described below is used by the client in issuing a ReadJournal request. It is received by the server when a ReadJournal indication 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.

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.

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.

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**.

`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 *)));
```

### JREAD_RESP_INFO

The operation-specific data structure described below is used by the server in issuing a ReadJournal response. The client receives it when a ReadJournal confirm 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.

---

***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);
```

## *ReadJournal Functions*

### u_mvl_jread_ind

**Usage:** This is a user defined function called when a ReadJournal indication is received. The user must examine the request parameters contained in the **`MVL_IND_PEND`** structure, do whatever is necessary to process the request, and then call **`mplas_jread_resp`** or **`mvlas_jread_resp`** to send the response (or **`mplas_err_resp`** to send an error response).

**Function Prototype:** `ST_VOID u_mvl_jread_ind (MVL_IND_PEND *ind_pend);`

**Parameters:**

`ind_pend` This is the same parameter that is passed to all user defined Indication functions. It contains a union of request and/or response parameters that is used for several different services. The request parameters are set by MVL before calling this function. The response parameters must be set by the user before calling the response function (i.e., **`mvlas_jread_resp`** or **`mplas_jread_resp`**). The parameters to be used for this service are as follows:

**Request parameters:**

| | |
|---|---|
| `ind_pend->u.jread.req_info` | See **`JREAD_REQ_INFO`** for more information. |

**Response parameters:**

| | |
|---|---|
| `ind_pend->u.jread.resp_info` | See **`JREAD_RESP_INFO`** for more information. |

**Return Value:** `ST_VOID`

## mplas_jread_resp

**Usage:** This function encodes and sends the Response for a previously received ReadJournal indication. The Response parameters in the **MVL_IND_PEND** structure must be filled in before this function is called. The (**MVL_IND_PEND ***) argument passed to this function must be the same as the (**MVL_IND_PEND ***) argument passed to the user defined function **u_mvl_jread_ind** when the indication was received.

**Function Prototype:** `ST_VOID mplas_jread_resp (MVL_IND_PEND *ind_pend);`

**Parameters:**

`ind_pend` This is the same parameter that is passed to all user defined Indication functions. It contains a union of request and/or response parameters that is used for several different services. The request parameters are set by MVL before calling this function. The response parameters must be set by the user before calling the response function (i.e., **mvlas_jread_resp** or **mplas_jread_resp**). The parameters to be used for this service are as follows:

**Request parameters:**

| | |
|---|---|
| `ind_pend->u.jread.req_info` | See **JREAD_REQ_INFO** for more information. |

**Response parameters:**

| | |
|---|---|
| `ind_pend->u.jread.resp_info` | See **JREAD_RESP_INFO** for more information. |

**Return Value:** `ST_VOID`

## mvlas_jread_resp

**Usage:** This is a Virtual Machine response function for handling a previously received ReadJournal indication. It completely processes the indication, fills in the response parameters in the **`MVL_IND_PEND`** structure, and then sends the response by calling **`mplas_jread_resp`**. The (**`MVL_IND_PEND *`**) argument passed to this function must be the same as the (**`MVL_IND_PEND *`**) argument passed to the user defined function **`u_mvl_jread_ind`** when the indication was received.

**Function Prototype:** `ST_VOID mvlas_jread_resp (MVL_IND_PEND *ind_pend);`

**Parameters:**

`ind_pend` This is the same parameter that is passed to all user defined Indication functions. It contains a union of request and/or response parameters that is used for several different services. The request parameters are set by MVL before calling this function. The response parameters must be set by the user before calling the response function (i.e., **`mvlas_jread_resp`** or **`mplas_jread_resp`**). The parameters to be used for this service are as follows:

**Request parameters:**

| | |
|---|---|
| `ind_pend->u.jread.req_info` | See **`JREAD_REQ_INFO`** for more information. |

**Response parameters:**

| | |
|---|---|
| `ind_pend->u.jread.resp_info` | See **`JREAD_RESP_INFO`** for more information. |

**Return Value:** `ST_VOID`

# ReportJournalStatus Service

This service is used to determine the number of entries in a Journal object.

## *ReportJournalStatus Data Structures*

The following is the ReportJournalStatus Indication Control Structure:

```
typedef struct
  {
  JSTAT_REQ_INFO *req_info;
  JSTAT_RESP_INFO *resp_info;
  } MVLAS_JSTAT_CTRL;
```

### JSTAT_REQ_INFO

The operation-specific structure described below is used by the client in issuing a ReportJournalStatus request. It is received by the server when a ReportJournalStatus indication 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. |

### JSTAT_RESP_INFO

The operation-specific data structure described below is used by the server in issuing a ReportJournalStatus response. It is received by the client when a ReportJournalStatus confirm 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.<br>**`SD_TRUE`**. This Journal is deletable using a service request. |

## *ReadJournalStatus Functions*

### u_mvl_jstat_ind

**Usage:** This is a user defined function called when a ReadJournalStatus indication is received. The user must examine the request parameters contained in the **`MVL_IND_PEND`** structure, do whatever is necessary to process the request, fill in the response parameters in the **`MVL_IND_PEND`** structure, and then call **`mplas_jstat_resp`** to send the response (or **`mplas_err_resp`** to send an error response).

**Function Prototype:** `ST_VOID u_mvl_jstat_ind (MVL_IND_PEND *ind_pend);`

**Parameters:**

`ind_pend` This is the same parameter that is passed to all user defined Indication functions. It contains a union of request and/or response parameters that is used for several different services. The request parameters are set by MVL before calling this function. The response parameters must be set by the user before calling the response function (i.e., **`mplas_jstat_resp`**). The parameters to be used for this service are as follows:

**Request parameters:**

| | |
|---|---|
| `ind_pend->u.jstat.req_info` | See **`JSTAT_REQ_INFO`** for more information. |

**Response parameters:**

| | |
|---|---|
| `ind_pend->u.jstat.resp_info` | See **`JSTAT_RESP_INFO`** for more information. |

**Return Value:** `ST_VOID`

## mplas_jstat_resp

**Usage:** This function encodes and sends the Response for a previously received ReadJournalStatus indication. The Response parameters in the **MVL_IND_PEND** structure must be filled in before this function is called. The (**MVL_IND_PEND ***) argument passed to this function must be the same as the (**MVL_IND_PEND ***) argument passed to the user defined function **u_mvl_jstat_ind** when the indication was received.

---

**Function Prototype:** `ST_VOID mplas_jstat_resp (MVL_IND_PEND *ind_pend);`

---

**Parameters:**

`ind_pend` This is the same parameter that is passed to all user defined Indication functions. It contains a union of request and/or response parameters that is used for several different services. The request parameters are set by MVL before calling this function. The response parameters must be set by the user before calling the response function (i.e., **mplas_jstat_resp**). The parameters to be used for this service are as follows:

**Request parameters:**

| | |
|---|---|
| `ind_pend->u.jstat.req_info` | See **JSTAT_REQ_INFO** for more information. |

**Response parameters:**

| | |
|---|---|
| `ind_pend->u.jstat.resp_info` | See **JSTAT_RESP_INFO** for more information. |

---

**Return Value:** `ST_VOID`

# ObtainFile Service

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.

## *ObtainFile Data Structures*

The following is the ObtainFile Indication Control Structure:

```
typedef struct
  {
  ST_CHAR srcfilename [MAX_FILE_NAME+1];
  ST_CHAR destfilename [MAX_FILE_NAME+1];
  } MVLAS_OBTFILE_CTRL;
```

**Fields**:

| | |
|---|---|
| `srcfilename` | Name of the source file. |
| `destfilename` | Name of the destination file. |

## *ObtainFile Functions*

### u_mvl_obtfile_ind

**Usage:** This is a user defined function called when a ObtainFile indication is received. The user must examine the request parameters contained in the **`MVL_IND_PEND`** structure, do whatever is necessary to process the request, and then call **`mplas_obtfile_resp`** or **`mvlas_obtfile_resp`** to send the response (or **`mplas_err_resp`** to send an error response). The application is responsible for issuing all the FileOpen, FileRead, and FileClose requests necessary to obtain the file before sending the ObtainFile response. **`mvlas_obtfile_resp`** takes care of the file transfer state machine and sending the response automatically.

**Function Prototype:** `ST_VOID u_mvl_obtfile_ind (MVL_IND_PEND *ind_pend);`

**Parameters:**

`ind_pend` This is the same parameter that is passed to all user defined Indication functions. It contains a union of request and/or response parameters that is used for several different services. The request parameters are set by MVL before calling this function. The parameters to be used for this service are as follows:

**Request parameters:**

`ind_pend->u.optfile.srcfilename`

`ind_pend->u.optfile.destfilename`

**Response parameters:**

NONE

**Return Value:** `ST_VOID`

## mplas_obtfile_resp

**Usage:** This function encodes and sends the Response for a previously received ObtainFile indication. The (**`MVL_IND_PEND *`**) argument passed to this function must be the same as the (**`MVL_IND_PEND *`**) argument passed to the user defined function **`u_mvl_obtfile_ind`** when the indication was received.

**Function Prototype:** `ST_VOID mplas_obtfile_resp (MVL_IND_PEND *ind_pend);`

**Parameters:**

`ind_pend` This is the same parameter that is passed to all user defined Indication functions. It contains a union of request and/or response parameters that is used for several different services. The request parameters are set by MVL before calling this function. The parameters to be used for this service are as follows:

**Request parameters:**

`ind_pend->u.optfile.srcfilename`

`ind_pend->u.optfile.destfilename`

**Response parameters:**

NONE

**Return Value:** `ST_VOID`

## mvlas_obtfile_resp

**Usage:** This 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_VOID mvlas_obtfile_resp (MVL_IND_PEND *ind_pend);`

**Parameters:**

`ind_pend` This is the same parameter that is passed to all user defined Indication functions. It contains a union of request and/or response parameters that is used for several different services. The request parameters are set by MVL before calling this function. The parameters to be used for this service are as follows:

**Request parameters:**

`ind_pend->u.optfile.srcfilename`

`ind_pend->u.optfile.destfilename`

**Response parameters:**

NONE

**Return Value:** `ST_VOID`

# FileRename Service

This service is used to rename a file on the VMD.

### *FileRename Data Structures*

The following is the FileRename Indication Control Structure:

```
typedef struct
  {
  ST_CHAR curfilename [MAX_FILE_NAME+1];
  ST_CHAR newfilename [MAX_FILE_NAME+1];
  } MVLAS_FRENAME_CTRL;
```

**Fields**:

`curfilename` This is a NULL terminated ASCII string that represents the current file name.

`newfilename` This is a NULL terminated ASCII string that represents the new file name.

### *FileRename Functions*

## u_mvl_frename_ind

**Usage:** This is a user defined function called when a FileRename indication is received. The user must examine the request parameters contained in the **`MVL_IND_PEND`** structure, do whatever is necessary to process the request, fill in the response parameters in the **`MVL_IND_PEND`** structure, and then call **`mplas_frename_resp`** to send the response (or **`mplas_err_resp`** to send an error response).

**Function Prototype:** `ST_VOID u_mvl_frename_ind (MVL_IND_PEND *ind_pend);`

**Parameters:**

`ind_pend` This is the same parameter that is passed to all user defined Indication functions. It contains a union of request and/or response parameters that is used for several different services. The request parameters are set by MVL before calling this function. The parameters to be used for this service are as follows:

**Request parameters:**

`ind_pend->u.frename.curfilename`

`ind_pend->u. frename.newfilename`

**Response parameters:**

NONE

**Return Value:** `ST_VOID`

## mplas_frename_resp

**Usage:** This function encodes and sends the Response for a previously received FileRename indication. There are no Response parameters in the **`MVL_IND_PEND`** structure to be filled in before this function is called but the application is responsible for renaming the file in the file store. The (**`MVL_IND_PEND *`**) argument passed to this function must be the same as the (**`MVL_IND_PEND *`**) argument passed to the user defined function **`u_mvl_frename_ind`** when the indication was received.

**Function Prototype:** `ST_VOID mplas_frename_resp (MVL_IND_PEND *ind);`

**Parameters:**

`ind_pend` This is the same parameter that is passed to all user defined Indication functions. It contains a union of request and/or response parameters that is used for several different services. The request parameters are set by MVL before calling this function. The parameters to be used for this service are as follows:

**Request parameters:**

`ind_pend->u.frename.curfilename`

`ind_pend->u.frename.newfilename`

**Response parameters:**

NONE

**Return Value:** `ST_VOID`

# 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.

## *FileOpen Data Structures*

The following is the FileOpen Indication Control Structure:

```
typedef struct
  {
  ST_CHAR filename [MAX_FILE_NAME+1];
  ST_INT init_pos;
  FOPEN_RESP_INFO *resp_info;
  } MVLAS_FOPEN_CTRL;
```

### FOPEN_RESP_INFO

The operation specific data structures described below are used by the server in issuing a FileOpen response. It is received by the client when a FileOpen confirm 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.

## *FileOpen Functions*

### u_mvl_fopen_ind

**Usage:** This is a user defined function called when a FileOpen indication is received. The user must examine the request parameters contained in the **`MVL_IND_PEND`** structure, do whatever is necessary to process the request, fill in the response parameters in the **`MVL_IND_PEND`** structure, and then call **`mplas_fopen_resp`** to send the response (or **`mplas_err_resp`** to send an error response).

**Function Prototype:** `ST_VOID u_mvl_fopen_ind (MVL_IND_PEND *ind_pend);`

**Parameters:**

`ind_pend`

This is the same parameter that is passed to all user defined Indication functions. It contains a union of request and/or response parameters that is used for several different services. The request parameters are set by MVL before calling this function. The response parameters must be set by the user before calling the response function (i.e., **`mplas_fopen_resp`**). The parameters to be used for this service are as follows:

**Request parameters:**

| | |
|---|---|
| `ind_pend->u.fopen.filename` | Name of file to open. |
| `ind_pend->u.fopen.init_pos` | Initial position in file. |

**Response parameters:**

| | |
|---|---|
| `ind_pend->u.fopen.resp_info` | See **`FOPEN_RESP_INFO`** for more information. |

**Return Value:** `ST_VOID`

## mplas_fopen_resp

**Usage:** This function encodes and sends the Response for a previously received FileOpen indication. The Response parameters in the **`MVL_IND_PEND`** structure must be filled in before this function is called. The (**`MVL_IND_PEND *`**) argument passed to this function must be the same as the (**`MVL_IND_PEND *`**) argument passed to the user defined function **`u_mvl_fopen_ind`** when the indication was received.

**Function Prototype:** `ST_VOID mplas_fopen_resp (MVL_IND_PEND *ind_pend);`

**Parameters:**

`ind_pend` This is the same parameter that is passed to all user defined Indication functions. It contains a union of request and/or response parameters that is used for several different services. The request parameters are set by MVL before calling this function. The response parameters must be set by the user before calling the response function (i.e., **`mplas_fopen_resp`**). The parameters to be used for this service are as follows:

**Request parameters:**

| | |
|---|---|
| `ind_pend->u.fopen.filename` | Name of file to open. |
| `ind_pend->u.fopen.init_pos` | Initial position in file. |

**Response parameters:**

| | |
|---|---|
| `ind_pend->u.fopen.resp_info` | See **`FOPEN_RESP_INFO`** for more information. |

**Return Value:** `ST_VOID`

# 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.

## *FileRead Data Structures*

The following is the FileRead Indication Control Structure:

```
typedef struct
  {
  FREAD_REQ_INFO  *req_info;
  ST_INT max_size;
  FREAD_RESP_INFO *resp_info;
  } MVLAS_FREAD_CTRL;
```

### FREAD_REQ_INFO

The operation specific data structure described below is used by the client in issuing the FileRead request. It is received by the server when a FileRead indication 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. |

### FREAD_RESP_INFO

The operation specific data structure described below is used by the server in issuing a FileRead response. It is received by the client when a FileRead confirm 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.<br>**`SD_FALSE`**. End-Of-File. No more data available. |

## *FileRead Functions*

## u_mvl_fread_ind

**Usage:** This is a user defined function called when a FileRead indication is received. The user must examine the request parameters contained in the **`MVL_IND_PEND`** structure, do whatever is necessary to process the request, fill in the response parameters in the **`MVL_IND_PEND`** structure, and then call **`mplas_fread_resp`** to send the response (or **`mplas_err_resp`** to send an error response).

**Function Prototype:** `ST_VOID u_mvl_fread_ind (MVL_IND_PEND *ind_pend);`

**Parameters:**

`ind_pend`

This is the same parameter that is passed to all user defined Indication functions. It contains a union of request and/or response parameters that is used for several different services. The request parameters are set by MVL before calling this function. The response parameters must be set by the user before calling the response function (i.e., **`mplas_fread_resp`**). The parameters to be used for this service are as follows:

**Request parameters:**

| | |
|---|---|
| `ind_pend->u.fread.req_info` | See **`FREAD_REQ_INFO`** for more information. |

**Response parameters:**

| | |
|---|---|
| `ind_pend->u.fread.resp_info` | See **`FREAD_RESP_INFO`** for more information. |

**Return Value:** `ST_VOID`

## mplas_fread_resp

**Usage:** This function encodes and sends the Response for a previously received FileRead indication. The Response parameters in the **`MVL_IND_PEND`** structure must be filled in before this function is called. The (**`MVL_IND_PEND *`**) argument passed to this function must be the same as the (**`MVL_IND_PEND *`**) argument passed to the user defined function **`u_mvl_fread_ind`** when the indication was received.

**Function Prototype:** `ST_VOID mplas_fread_resp (MVL_IND_PEND *ind_pend);`

**Parameters:**

`ind_pend` This is the same parameter that is passed to all user defined Indication functions. It contains a union of request and/or response parameters that is used for several different services. The request parameters are set by MVL before calling this function. The response parameters must be set by the user before calling the response function (i.e., **`mplas_fread_resp`**). The parameters to be used for this service are as follows:

**Request parameters:**

| | |
|---|---|
| `ind_pend->u.fread.req_info` | See **`FREAD_REQ_INFO`** for more information. |

**Response parameters:**

| | |
|---|---|
| `ind_pend->u.fread.resp_info` | See **`FREAD_RESP_INFO`** for more information. |

**Return Value:** `ST_VOID`

# 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.

## *FileClose Data Structures*

The following is the FileClose Indication Control Structure:

```
typedef struct
  {
  FCLOSE_REQ_INFO *req_info;
  } MVLAS_FCLOSE_CTRL;
```

### FCLOSE_REQ_INFO

The operation specific data structure described below is used by the client in issuing the FileClose request. It is received by the server when a FileClose indication 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`**. |

## *FileClose Functions*

## u_mvl_fclose_ind

**Usage:** This is a user defined function called when a FileClose indication is received. The user must examine the request parameters contained in the **`MVL_IND_PEND`** structure, do whatever is necessary to process the request, fill in the response parameters in the **`MVL_IND_PEND`** structure, and then call **`mplas_fclose_resp`** to send the response (or **`mplas_err_resp`** to send an error response).

**Function Prototype:** `ST_VOID u_mvl_fclose_ind (MVL_IND_PEND *ind_pend);`

**Parameters:**

`ind_pend` This is the same parameter that is passed to all user defined Indication functions. It contains a union of request and/or response parameters that is used for several different services. The request parameters are set by MVL before calling this function. The parameters to be used for this service are as follows:

**Request parameters:**

| | |
|---|---|
| `ind_pend->u.fclose.req_info` | See **`FCLOSE_REQ_INFO`** for more information. |

**Response parameters:**

NONE

**Return Value:** `ST_VOID`

## mplas_fclose_resp

**Usage:** This function encodes and sends the Response for a previously received FileClose indication. The (**`MVL_IND_PEND *`**) argument passed to this function must be the same as the (**`MVL_IND_PEND *`**) argument passed to the user defined function **`u_mvl_fclose_ind`** when the indication was received.

**Function Prototype:** `ST_VOID mplas_fclose_resp (MVL_IND_PEND *ind_pend);`

**Parameters:**

`ind_pend` This is the same parameter that is passed to all user defined Indication functions. It contains a union of request and/or response parameters that is used for several different services. The request parameters are set by MVL before calling this function. The parameters to be used for this service are as follows:

**Request parameters:**

| | |
|---|---|
| `ind_pend->u.fclose.req_info` | See **`FCLOSE_REQ_INFO`** for more information. |

**Response parameters:**

NONE

**Return Value:** `ST_VOID`

# FileDelete Service

This service is used by a client to delete a file from the virtual filestore of a server.

### FileDelete Data Structures

The following is the FileDelete Indication Control Structure:

```
typedef struct
  {
  ST_CHAR filename [MAX_FILE_NAME+1];
  } MVLAS_FDELETE_CTRL;
```

## FileDelete Functions

### u_mvl_fdelete_ind

**Usage:** This is a user defined function called when a FileDelete indication is received. The user must examine the request parameters contained in the **`MVL_IND_PEND`** structure, do whatever is necessary to process the request, and then call **`mplas_fdelete_resp`** to send the response (or **`mplas_err_resp`** to send an error response).

**Function Prototype:** `ST_VOID u_mvl_fdelete_ind (MVL_IND_PEND *ind_pend);`

**Parameters:**

`ind_pend` This is the same parameter that is passed to all user defined Indication functions. It contains a union of request and/or response parameters that is used for several different services. The request parameters are set by MVL before calling this function. The parameters to be used for this service are as follows:

**Request parameters:**

| | |
|---|---|
| `ind_pend->u.fdelete.filename` | Name of file to open. |

**Response parameters:**

NONE

**Return Value:** `ST_VOID`

## mplas_fdelete_resp

**Usage:** This function encodes and sends the Response for a previously received FileDelete indication. The (**MVL_IND_PEND ***) argument passed to this function must be the same as the (**MVL_IND_PEND ***) argument passed to the user defined function **u_mvl_fdelete_ind** when the indication was received.

**Function Prototype:** ST_VOID mplas_fdelete_resp (MVL_IND_PEND *ind_pend);

**Parameters:**

ind_pend

This is the same parameter that is passed to all user defined Indication functions. It contains a union of request and/or response parameters that is used for several different services. The request parameters are set by MVL before calling this function. The parameters to be used for this service are as follows:

**Request parameters:**

| | |
|---|---|
| ind_pend->u.fdelete.filename | Name of file to open. |

**Response parameters:**

NONE

**Return Value:** ST_VOID

# 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 filestore. The attributes returned by this service are the same as those returned in the FileOpen service.

## *FileDirectory Data Structures*

The following is the FileDirectory Indication Control Structure:

```
typedef struct
  {
  ST_CHAR fs_filename[MAX_FILE_NAME+1];
  ST_CHAR ca_filename[MAX_FILE_NAME+1];
  MVL_FDIR_RESP_INFO *resp_info;
  } MVLAS_FDIR_CTRL;
```

### MVL_DIR_ENT

```
typedef struct
  {
  ST_UINT32 fsize;                    /* file size (# bytes)        */
  ST_BOOLEAN mtimpres;                /* last modified time present */
  time_t mtime;                       /* last modified time         */
  ST_CHAR filename [MAX_FILE_NAME+1];
  } MVL_DIR_ENT;
```

### MVL_FDIR_RESP_INFO

```
typedef struct
  {
  ST_INT num_dir_ent;         /* number of directory entries      */
  ST_BOOLEAN more_follows;    /* more dir entries follow          */
                              /* default: SD_FALSE                */
  MVL_DIR_ENT *dir_ent;       /* ptr to array of dir entries      */
  } MVL_FDIR_RESP_INFO;
```

## *FileDirectory Functions*

### u_mvl_fdir_ind

**Usage:** This is a user defined function called when a FileDirectory indication is received. The user must examine the request parameters contained in the **MVL_IND_PEND** structure, do whatever is necessary to process the request, fill in the response parameters in the **MVL_IND_PEND** structure, and then call **mplas_fdir_resp** to send the response (or **mplas_err_resp** to send an error response).

**Function Prototype:** `ST_VOID u_mvl_fdir_ind (MVL_IND_PEND *ind_pend);`

**Parameters:**

`ind_pend` This is the same parameter that is passed to all user defined Indication functions. It contains a union of request and/or response parameters that is used for several different services. The request parameters are set by MVL before calling this function. The response parameters must be set by the user before calling the response function (i.e., **mplas_fdir_resp**). The parameters to be used for this service are as follows:

**Request parameters:**

| | |
|---|---|
| `ind_pend->u.fdir.fs_filename` | File Specification. |
| `ind_pend->u.fdir.ca_filename` | File name to continue after. |

**Response parameters:**

| | |
|---|---|
| `ind_pend->u.fdir.resp_info` | See **MVL_FDIR_RESP_INFO** for more information. |

**Return Value:** `ST_VOID`

## mplas_fdir_resp

**Usage:** This function encodes and sends the Response for a previously received FileDirectory indication. The Response parameters in the **`MVL_IND_PEND`** structure must be filled in before this function is called. The (**`MVL_IND_PEND *`**) argument passed to this function must be the same as the (**`MVL_IND_PEND *`**) argument passed to the user defined function **`u_mvl_fdir_ind`** when the indication was received.

**Function Prototype:** `ST_VOID mplas_fdir_resp (MVL_IND_PEND *ind_pend);`

**Parameters:**

`ind_pend` This is the same parameter that is passed to all user defined Indication functions. It contains a union of request and/or response parameters that is used for several different services. The request parameters are set by MVL before calling this function. The response parameters must be set by the user before calling the response function (i.e., **`mplas_fdir_resp`**). The parameters to be used for this service are as follows:

**Request parameters:**

| | |
|---|---|
| `ind_pend->u.fdir.fs_filename` | File Specification. |
| `ind_pend->u.fdir.ca_filename` | File name to continue after. |

**Response parameters:**

| | |
|---|---|
| `ind_pend->u.fdir.resp_info` | See **`MVL_FDIR_RESP_INFO`** for more information. |

**Return Value:** `ST_VOID`

# MVL MMS Client Facilities

MVL provides high-level client functions for many MMS services. Both synchronous (with timeout) and asynchronous versions of all client services are available. Please see the MVL sample client source code in **client.c** for more detail.

MVL has an outstanding request control system that keeps track of requests and matches up responses. The maximum number of outstanding request control elements that it will allocate is set by the global variable **mvl_max_req_pend** (default value is 10).

***Note:*** *This is a list based system designed to handle multiple outstanding requests, and a more efficient system that allows a single outstanding request is also included and is compiled in by deleting the define **ALLOW_MULTIPLE_REQUESTS_OUT**. The simple implementation will suffice for simple clients or those that use only the synchronous request functions, and will save a bit of code space.*

*Also, note that MVL allows the user to configure the remote node's variables, variable lists, and domains as well. This information is then used to handle Information Reports, and pre/post processing is available for these objects as well.*

If the user application makes use of asynchronous client request functions (such as **mvla_read_variables**), the **u_req_done** callback function from the **MMS_REQ_PEND** structure will be invoked (if not **NULL**) from within the **mvl_comm_serve** function.

## General Data Structure

### MVL_REQ_PEND

The Client uses this structure for tracking any outstanding request. It contains all the information needed to match up a response with a request. When a response is received, it also contains all the necessary Response information.

```
typedef struct mvl_req_pend
  {
  DBL_LNK l;                        /* For linked list            */
  ST_UINT32 invoke_id;
  time_t request_time;
  MVL_NET_INFO *net_info;           /* Who it was sent to         */
  ST_INT op;                        /* MMS Opcode                 */
  union
    {
    struct
      {
      MVL_READ_RESP_PARSE_INFO *parse_info;
      ST_INT num_data;
      } rd;
    struct
      {
      MVL_WRITE_REQ_INFO *w_info;
      ST_INT num_data;
      } wr;
    struct
      {
      IDENT_RESP_INFO *resp_info;
      } ident;
    struct
      {
      INIT_INFO *resp_info;
      } init;
    struct
      {
      FOPEN_RESP_INFO *resp_info;
      } fopen;
    struct
      {
      FREAD_RESP_INFO *resp_info;
      } fread;
    struct
      {
      MVL_FDIR_RESP_INFO *resp_info;
      } fdir;
    struct
      {
      JINIT_RESP_INFO *resp_info;
      } jinit;
    struct
      {
      JSTAT_RESP_INFO *resp_info;
      } jstat;
    struct
      {
      MVL_JREAD_RESP_INFO *resp_info;
      } jread;
    struct
      {
      GETVLIST_RESP_INFO *resp_info;
      } getvlist;
    struct
      {
      DELVLIST_RESP_INFO *resp_info;
      } delvlist;
    } u;
  MVL_COMM_EVENT *event;   /* Save event ptr to free later           */
  ST_BOOLEAN done;
  ST_RET result;           /* SD_SUCCESS or error code               */
                           /* User done function for async functions */
  ST_VOID (*u_req_done) (struct mvl_req_pend *req);

  ST_VOID *v;              /* For MVL user's use                     */
  } MVL_REQ_PEND;
```

# Client Support Functions

The following functions are used for all Client services.

## mvl_free_req_ctrl

**Usage:** Every Client request function allocates a **`MVL_REQ_PEND`** structure for tracking the request and to hold the response information. This function must be called to free the structure sometime after the response is received and processed.

***IMPORTANT*:** After this function is called, the structure contents are no longer valid. The contents of the structure MUST NOT be used after this function is called. This applies to any pointers to response information (i.e., **`req_pend->u.ident.resp_info`**). If some of the response information is still needed after calling this function, it must be copied to a safe location before calling this function.

**Function Prototype:** `ST_VOID mvl_free_req_ctrl (MVL_REQ_PEND *req_pend);`

**Parameters:**

`req_pend` Pointer to request tracking structure returned from a call to a Client request function (i.e., If **`&req_pend`** was passed as the **`req_out`** parameter to the Client request function then **`req_pend`** should be passed to this function).

**Return Value:** `ST_VOID`

## u_mvl_check_timeout

**Usage:** This function pointer may be set to point to a user-defined function called repetitively by synchronous MVL client request functions while waiting for a confirm. If this function returns **`SD_TRUE`**, the synchronous request function will stop waiting for the confirm, and will return immediately with an error code. The user-defined function can be used to perform other applications processing, but cannot be used to perform additional MMS-EASE *Lite* communications activity.

**Function Pointer Global Variable:**

```
extern ST_BOOLEAN (*u_mvl_check_timeout)
                  (ST_VOID);
```

**Parameters:** NONE

| | | | |
|---|---|---|---|
| **Return Value:** | `ST_BOOLEAN` | `SD_TRUE` | Stop waiting for Confirm. |
| | | `SD_FALSE` | Continue waiting for Confirm. |

# Client Request Functions Overview

Each Client service may be processed synchronously or asynchronously, simply by calling a different request function.

## *Synchronous Request Functions*

All of the synchronous request functions begin with the "**`mvl_`**" prefix. The synchronous request functions do not return until the response has been received (or a timeout occurs).

## *Asynchronous Request Functions*

All of the asynchronous request functions begin with the "**`mvla_`**" prefix. The asynchronous functions return immediately. To determine when the response has been received, the user can check the value of **`done`** in the **`MVL_REQ_PEND`** structure. If it is **`SD_FALSE`**, the response has NOT been received yet. Alternatively, the user can set the function pointer **`u_req_done`** in the **`MVL_REQ_PEND`** structure to point to a function that will be called when the response is received. In either case, when the response is received, the user must check the value of **`result`** in the **`MVL_REQ_PEND`** structure to determine if the request was successful or not.

## *Common Arguments to Request Functions*

The following arguments are passed to all Client request functions (synchronous and asynchronous).

`net_info` — This argument indicates where to send the request (i.e., which network connection to use).

`req_out` — Every request function (synchronous and asynchronous) includes an argument **`MVL_REQ_PEND *req_out`**. The user must pass the address of a variable of type (**`MVL_REQ_PEND *`**) to the function. The function allocates a **`MVL_REQ_PEND`** structure and sets the user's variable to the address of the allocated structure. For example, if the user has a variable **`MVL_REQ_PEND *req_pend`**, they should pass **`&req_pend`** to the function and it will set the value of **`req_pend`**. The user must free the structure sometime after the response is received and processed by calling **`mvl_free_req_ctrl`** (**`req_pend`**).

# 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;
```

<u>**Fields**</u>:

`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,
   utc-time         [17] IMPLICIT UtcTime
}
```

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.

### 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.

## *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`**.*

## *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 structure 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. |

# 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.

## *Read Data Structures*

### READ_REQ_INFO

The operation-specific data structure described below is used by the Client in issuing the variable read request function. It is received by the Server when a variable read indication function 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.<br>**`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. |

***Note:*** *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)));
```

### MVL_READ_RESP_PARSE_INFO

This structure contains information for processing the Read Response Data.

```
typedef struct mvl_read_resp_parse_info
  {
  ST_RET result;          /* SD_SUCCESS for OK                      */
  ST_VOID *dest;          /* Where data is to be put                */
  ST_INT type_id;         /* type of variable                       */
  ST_INT descr_arr;       /* for described read of array            */
  ST_INT arr_size;        /* number of elements in described array. */
                          /* Used only if descr_arr != SD_FALSE     */
  } MVL_READ_RESP_PARSE_INFO;
```

## *Read Functions*

## mvl_read_variables

**Usage:** This function performs a synchronous Read request.

**Function Prototype:**

```
ST_RET mvl_read_variables (MVL_NET_INFO *net_info,
                           READ_REQ_INFO *read_info,
                           ST_INT num_data,
                           MVL_READ_RESP_PARSE_INFO *parse_info,
                           MVL_REQ_PEND **req_out);
```

**Parameters:**

| | |
|---|---|
| `net_info` | Network connection information. |
| `read_info` | Read request information. |
| `num_data` | Number of variables to read. |
| `parse_info` | Pointer to array of structures, one for each variable. Each structure contains the information necessary for processing the response data for a single variable. The parameters **`dest`**, **`type_id`**, **`descr_arr`**, and optionally **`arr_size`** must be set before calling this function. The **`result`** parameter is set by MVL when the response is received. |
| `req_out` | See the description of **`req_out`** on page 183. |

**Response Data:** If **`i`** is the index into the list of variables, then:

| | |
|---|---|
| `parse_info [i].result` | Indicates if the variable was read successfully. |
| `parse_info [i].dest` | Contains the value of the variable. |

| | | | |
|---|---|---|---|
| **Return Value:** | `ST_RET` | `SD_SUCCESS` | If request sent and response received successfully, or error code. |

## mvla_read_variables

**Usage:** This function performs an asynchronous Read request.

**Function Prototype:**

```
ST_RET mvla_read_variables (MVL_NET_INFO  *net_info,
                            READ_REQ_INFO *read_info,
                            ST_INT num_data,
                            MVL_READ_RESP_PARSE_INFO *parse_info,
                            MVL_REQ_PEND **req_out);
```

**Parameters:**

| | |
|---|---|
| `net_info` | Network connection information |
| `read_info` | Read request information |
| `num_data` | Number of variables to read |
| `parse_info` | Pointer to array of structures, one for each variable. Each structure contains the information necessary for processing the response data for a single variable. The parameters **`dest`**, **`type_id`**, **`descr_arr`**, and optionally **`arr_size`** must be set before calling this function. The **`result`** parameter is set by MVL when the response is received. |
| `req_out` | See the description of **`req_out`** on page 183. |

**Response Data:** If **`i`** is the index into the list of variables, then:

| | |
|---|---|
| `parse_info [i].result` | Indicates if the variable was read successfully. |
| `parse_info [i].dest` | Contains the value of the variable. |

**Return Value:**

| | | |
|---|---|---|
| `ST_RET` | `SD_SUCCESS` | If request sent successfully, or error code. |

**Note:** If the return value is **`SD_SUCCESS`**, you still need to wait for the response. See *Asynchronous Request Functions* on page 183 for information on how to wait.

# 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.

## *Write Data Structures*

### WRITE_REQ_INFO

This operation-specific data structure described below is used by the Client in issuing a variable write request. It is received by the Server when a variable write indication 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.

***Note:*** *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)));
```

### MVL_WRITE_REQ_INFO

This structure contains request and response parameters. See the function description for how they are used.

```
typedef struct mvl_write_req_info
  {
  ST_RET result;           /* SD_SUCCESS for OK                    */
  ST_VOID *local_data;     /* Source of local data                 */
  ST_INT local_data_size;  /* Size of local data                   */
  ST_INT type_id;          /* type of variable                     */
  ST_BOOLEAN arr;          /* SD_TRUE if type is array and the #  */
  ST_INT num_el;           /* elements needs to be set             */
  } MVL_WRITE_REQ_INFO;
```

## *Write Functions*

## mvl_write_variables

**Usage:** This function performs a synchronous Write request.

**Function Prototype:**

```
ST_RET mvl_write_variables (MVL_NET_INFO *net_info,
                            WRITE_REQ_INFO *write_info,
                            ST_INT num_data,
                            MVL_WRITE_REQ_INFO *w_info,
                            MVL_REQ_PEND *req_out);
```

**Parameters:**

| | |
|---|---|
| `net_info` | Network connection information. |
| `write_info` | Write request information. |
| `num_data` | Number of variables to write. |
| `w_info` | Pointer to array of structures, one for each variable. Each structure contains the information about the data to be written. The parameters **`local_data`**, **`local_data_size`**, **`type_id`**, **`arr`**, and optionally **`num_el`** must be set before calling this function. The **`result`** parameter is set by MVL when the response is received. |
| `req_out` | See the description of **`req_out`** on page 183. |

**Response Data:** If “i” is the index into the list of variables, then:

| | |
|---|---|
| `w_info [i].result` | Indicates if the variable was written successfully. |

**Return Value:**

| | | |
|---|---|---|
| `ST_RET` | `SD_SUCCESS` | If request sent and response received successfully, or error code. |

## mvla_write_variables

**Usage:** This function performs an asynchronous Write request.

**Function Prototype:**

```
ST_RET mvla_write_variables (MVL_NET_INFO *net_info,
                             WRITE_REQ_INFO *write_info,
                             ST_INT num_data,
                             MVL_WRITE_REQ_INFO *w_info,
                             MVL_REQ_PEND *req_out);
```

**Parameters:**

| | |
|---|---|
| `net_info` | Network connection information. |
| `write_info` | Write request information. |
| `num_data` | Number of variables to write. |
| `w_info` | Pointer to array of structures, one for each variable. Each structure contains the information about the data to be written. The parameters **`local_data`**, **`local_data_size`**, **`type_id`**, **`arr`**, and optionally **`num_el`** must be set before calling this function. The **`result`** parameter is set by MVL when the response is received. |
| `req_out` | See the description of **`req_out`** on page 183. |

**Response Data:** If “i” is the index into the list of variables, then:

| | |
|---|---|
| `w_info [i].result` | Indicates if the variable was written successfully. |

**Return Value:**

| | | |
|---|---|---|
| `ST_RET` | `SD_SUCCESS` | If request sent successfully, or error code. |

**Note:** If the return value is **`SD_SUCCESS`**, you still need to wait for the response. See *Asynchronous Request Functions* on page 183 for information on how to wait.

# 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.

## *InformationReport Functions*

### u_mvl_info_rpt_ind

**Usage:** This is a user defined function called when an InformationReport indication is received. The user may examine the data referenced by the **`MVL_COMM_EVENT`** structure. Because this is an unconfirmed service, there is no response to send.

**Function Prototype:** `ST_VOID u_mvl_info_rpt_ind (MVL_COMM_EVENT *event);`

**Parameters:**

`event` This is a pointer to a structure containing all the information from the request. The structure **`MVL_COMM_EVENT`** is defined in **mvl_defs.h**.

**Return Value:** `ST_VOID`

**Note:** An example of this user defined function can be found in **client.c**. It may be convenient to make use of the **`mvl_info_data_to_local`** function to convert the data to local format.

## mvl_info_data_to_local

**Usage:** This function converts InformationReport data to local format. The user must provide an array of pointers to Variable Association structures (**MVL_VAR_ASSOC**). If you are processing both received IEC-61850 and UCA reports, you must call this function more than once with different values in the **num_va** argument. This is documented in the client code, **cli_rpt.c**.

**Function Prototype:**

```
ST_VOID mvl_info_data_to_local (MVL_COMM_EVENT *event,
                                ST_INT num_va,
                                MVL_VAR_ASSOC **info_va);
```

**Parameters:**

| | |
|---|---|
| event | This is a pointer to a structure containing all the information from the request. The structure **MVL_COMM_EVENT** is defined in **mvl_defs.h**. |
| num_va | Number of variables to convert to local format. |
| info_va | Pointer to array of pointers to Variable Association structures. These structures must contain valid data type information (i.e., **info_va[i].type_id**) to be used in the conversion to local format, and valid pointers to data buffers (i.e., **info_va[i].data**) where the data can be stored. |

**Return Value:** ST_VOID

# Status Service

This service is used to allow a client to determine the general condition or status of a server node.

## *Status Data Structures*

### STATUS_REQ_INFO

See page 94 for more information.

### STATUS_RESP_INFO

See page 94 for more information.

## *Status Functions*

## mvl_status

**Usage:** This function performs a synchronous Status request.

**Function Prototype:**

```
ST_RET mvl_status (MVL_NET_INFO  *net_info,
                   STATUS_REQ_INFO *req_info,
                   MVL_REQ_PEND **req_out);
```

**Parameters:**

| | |
|---|---|
| `net_info` | Network connection information. |
| `req_info` | Status request information. |
| `req_out` | See the description of **`req_out`** on page 183. |

**Response Data:** The response data is in the `status` parameter of the **`MVL_REQ_PEND`** structure. It may be referenced by the following statement (assuming **`&req_pend`** was passed as the **`req_out`** argument):

```
STATUS_RESP_INFO *resp_info = req_pend->u.status.resp_info;
```

**Return Value:**

| | | | |
|---|---|---|---|
| **Return Value:** | `ST_RET` | `SD_SUCCESS` | If request sent and response received successfully, or error code. |

## mvla_status

**Usage:** This function performs an asynchronous Status request.

**Function Prototype:**

```
ST_RET mvla_status (MVL_NET_INFO  *net_info,
                    STATUS_REQ_INFO *req_info,
                    MVL_REQ_PEND **req_out);
```

**Parameters:**

| | |
|---|---|
| `net_info` | Network connection information. |
| `req_info` | Status request information. |
| `req_out` | See the description of **`req_out`** on page 183. |

**Response Data:** The response data is in the **`status`** parameter of the **`MVL_REQ_PEND`** structure. It may be referenced by the following statement (assuming **`&req_pend`** was passed as the **`req_out`** argument):

```
STATUS_RESP_INFO *resp_info = req_pend->u.status.resp_info;
```

**Return Value:** `ST_RET` `SD_SUCCESS` If request sent successfully, or error code.

**Note:** If the return value is **`SD_SUCCESS`**, you still need to wait for the response. See *Asynchronous Request Functions* on page 183 for information on how to wait.

# Identify Service

This service is used to obtain identifying information such a vendor name, and model number, from a responding node.

## *Identify Data Structures*

### IDENT_RESP_INFO

See page 97 for more information.

## *Identify Functions*

### mvl_identify

**Usage:** This function performs a synchronous Identify request.

**Function Prototype:**

```
ST_RET mvl_identify (MVL_NET_INFO *net_info,
                     MVL_REQ_PEND **req_out);
```

**Parameters:**

| | |
|---|---|
| `net_info` | Network connection information. |
| `req_out` | See the description of **`req_out`** on page 183. |

**Response Data:** The response data is in the **`ident`** parameter of the **`MVL_REQ_PEND`** structure. It may be referenced by the following statement (assuming **`&req_pend`** was passed as the **`req_out`** argument):

```
IDENT_RESP_INFO *ident = req_pend->u.ident.resp_info;
```

**Return Value:** `ST_RET` `SD_SUCCESS` If request sent and response received successfully, or error code.

## mvla_identify

**Usage:** This function performs an asynchronous Identify request.

**Function Prototype:**

```
ST_RET mvla_identify (MVL_NET_INFO  *net_info,
                      MVL_REQ_PEND **req_out);
```

| | |
|---|---|
| `net_info` | Network connection information. |
| `req_out` | See the description of **`req_out`** on page 183. |

**Response Data:** The response data is in the **`ident`** parameter of the `MVL_REQ_PEND` structure. It may be referenced by the following statement (assuming **`&req_pend`** was passed as the **`req_out`** argument):

```
IDENT_RESP_INFO *ident = req_pend->u.ident.resp_info;
```

**Return Value:** `ST_RET` `SD_SUCCESS` If request sent successfully, or error code.

**Note:** If the return value is **`SD_SUCCESS`**, you still need to wait for the response. See *Asynchronous Request Functions* on page 183 for information on how to wait.

# 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.

## *GetNameList Data Structures*

### NAMELIST_REQ_INFO

See page 100 for more information.

### NAMELIST_RESP_INFO

See page 102 for more information.

## *GetNameList Functions*

### mvl_getnam

**Usage:** This function performs a synchronous GetNameList request.

**Function Prototype:**

```
ST_RET mvl_getnam (MVL_NET_INFO  *net_info,
                   NAMELIST_REQ_INFO *req_info,
                   MVL_REQ_PEND **req_out);
```

**Parameters:**

| | |
|---|---|
| `net_info` | Network connection information. |
| `req_info` | GetNameList request information. |
| `req_out` | See the description of **`req_out`** on page 183. |

**Response Data:** The response data is in the **`getnam`** parameter of the `MVL_REQ_PEND` structure. It may be referenced by the following statement (assuming **`&req_pend`** was passed as the **`req_out`** argument):

```
NAMELIST_RESP_INFO *resp_info = req_pend->u.getnam.resp_info;
```

**Return Value:** `ST_RET` `SD_SUCCESS` If request sent and response received successfully, or error code.

## mvla_getnam

**Usage:** This function performs an asynchronous GetNameList request.

**Function Prototype:**

```
ST_RET mvla_getnam (MVL_NET_INFO  *net_info,
                    NAMELIST_REQ_INFO *req_info,
                    MVL_REQ_PEND **req_out);
```

**Parameters:**

| | |
|---|---|
| net_info | Network connection information. |
| req_info | GetNameList request information. |
| req_out | See the description of **req_out** on page 183. |

**Response Data:** The response data is in the **getnam** parameter of the MVL_REQ_PEND structure. It may be referenced by the following statement (assuming **&req_pend** was passed as the **req_out** argument):

```
NAMELIST_RESP_INFO *resp_info = req_pend->u.getnam.resp_info;
```

**Return Value:** ST_RET SD_SUCCESS If request sent successfully, or error code.

**Note:** If the return value is **SD_SUCCESS**, you still need to wait for the response. See *Asynchronous Request Functions* on page 183 for information on how to wait.

# 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.

## *FileOpen Data Structures*

### FOPEN_RESP_INFO

See page 166 for more information.

## *FileOpen Functions*

## mvl_fopen

**Usage:** This function performs a synchronous FileOpen request.

**Function Prototype:**

```
ST_RET mvl_fopen (MVL_NET_INFO  *net_info,
                  ST_CHAR *filename,
                  ST_UINT32 init_pos,
                  MVL_REQ_PEND **req_out);
```

**Parameters:**

| | |
|---|---|
| `net_info` | Network connection information. |
| `filename` | Name of file to open (NULL-terminated string). |
| `init_pos` | Initial position in file to begin reading (i.e., number of bytes to skip). |
| `req_out` | See the description of **`req_out`** on page 183. |

**Response Data:** The response data is in the fopen parameter of the `MVL_REQ_PEND` structure. It may be referenced by the following statement (assuming **`&req_pend`** was passed as the **`req_out`** argument):

```
FOPEN_RESP_INFO *resp_info = req_pend->u.fopen.resp_info;
```

| | | | |
|---|---|---|---|
| **Return Value:** | `ST_RET` | `SD_SUCCESS` | If request sent and response received successfully, or error code. |

## mvla_fopen

**Usage:** This function performs an asynchronous FileOpen request.

**Function Prototype:**

```
ST_RET mvla_fopen (MVL_NET_INFO  *net_info,
                   ST_CHAR *filename,
                   ST_UINT32 init_pos,
                   MVL_REQ_PEND **req_out);
```

**Parameters:**

| | |
|---|---|
| `net_info` | Network connection information. |
| `filename` | Name of file to open (NULL-terminated string). |
| `init_pos` | Initial position in file to begin reading (i.e., number of bytes to skip). |
| `req_out` | See the description of **`req_out`** on page 183. |

**Response Data:** The response data is in the **`fopen`** parameter of the `MVL_REQ_PEND` structure. It may be referenced by the following statement (assuming **`&req_pend`** was passed as the **`req_out`** argument):

```
FOPEN_RESP_INFO *resp_info = req_pend->u.fopen.resp_info;
```

**Return Value:** `ST_RET` `SD_SUCCESS` If request sent successfully, or error code.

**Note:** If the return value is **`SD_SUCCESS`**, you still need to wait for the response. See *Asynchronous Request Functions* on page 183 for information on how to wait.

# 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.

## *FileRead Data Structures*

### FREAD_REQ_INFO

See page 169 for more information.

### FREAD_RESP_INFO

See page 169 for more information.

## *FileRead Functions*

## mvl_fread

**Usage:** This function performs a synchronous FileRead request.

**Function Prototype:**

```
ST_RET mvl_fread (MVL_NET_INFO  *net_info,
                  FREAD_REQ_INFO *req_info,
                  MVL_REQ_PEND **req_out);
```

**Parameters:**

| | |
|---|---|
| `net_info` | Network connection information. |
| `req_info` | FileRead request information. |
| `req_out` | See the description of **`req_out`** on page 183. |

**Response Data:** The response data is in the **`fread`** parameter of the `MVL_REQ_PEND` structure. It may be referenced by the following statement (assuming **`&req_pend`** was passed as the **`req_out`** argument):

```
FREAD_RESP_INFO *resp_info = req_pend->u.fread.resp_info;
```

**Return Value:**

| | | |
|---|---|---|
| `ST_RET` | `SD_SUCCESS` | If request sent and response received successfully, or error code. |

## mvla_fread

**Usage:** This function performs an asynchronous FileRead request.

**Function Prototype:**

```
ST_RET mvla_fread (MVL_NET_INFO  *net_info,
                   FREAD_REQ_INFO *req_info,
                   MVL_REQ_PEND **req_out);
```

**Parameters:**

| | |
|---|---|
| `net_info` | Network connection information. |
| `req_info` | FileRead request information. |
| `req_out` | See the description of **`req_out`** on page 183. |

**Response Data:** The response data is in the **`fread`** parameter of the `MVL_REQ_PEND` structure. It may be referenced by the following statement (assuming **`&req_pend`** was passed as the **`req_out`** argument):

```
FREAD_RESP_INFO *resp_info = req_pend->u.fread.resp_info;
```

**Return Value:** `ST_RET` `SD_SUCCESS` If request sent successfully, or error code.

**Note:** If the return value is **`SD_SUCCESS`**, you still need to wait for the response. See *Asynchronous Request Functions* on page 183 for information on how to wait.

# 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.

### *FileClose Data Structures*

#### FCLOSE_REQ_INFO

See page 172 for more information.

## *FileClose Functions*

## mvl_fclose

**Usage:** This function performs a synchronous FileClose request.

**Function Prototype:**

```
ST_RET mvl_fclose (MVL_NET_INFO  *net_info,
                   FCLOSE_REQ_INFO *req_info,
                   MVL_REQ_PEND **req_out);
```

**Parameters:**

| | |
|---|---|
| `net_info` | Network connection information. |
| `req_info` | FileClose request information. |
| `req_out` | See the description of **`req_out`** on page 183. |

| | | | |
|---|---|---|---|
| **Return Value:** | `ST_RET` | `SD_SUCCESS` | If request sent and response received successfully, or error code. |

## mvla_fclose

**Usage:** This function performs an asynchronous FileClose request.

**Function Prototype:**

```
ST_RET mvla_fclose (MVL_NET_INFO  *net_info,
                    FCLOSE_REQ_INFO *req_info,
                    MVL_REQ_PEND **req_out);
```

**Parameters:**

| | |
|---|---|
| `net_info` | Network connection information. |
| `req_info` | FileClose request information. |
| `req_out` | See the description of **`req_out`** on page 183. |

**Return Value:** `ST_RET` `SD_SUCCESS` If request sent successfully, or error code.

**Note:** If the return value is **`SD_SUCCESS`**, you still need to wait for the response. See *Asynchronous Request Functions* on page 183 for information on how to wait.

# 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 filestore. The attributes returned by this service are the same as those returned in the FileOpen service.

## *FileDirectory Data Structures*

### MVL_DIR_ENT

This structure contains data for a single **FileDirectory** entry (i.e., a single file).

```
typedef struct
  {
  ST_UINT32 fsize;                  /* file size (# bytes)        */
  ST_BOOLEAN mtimpres;              /* last modified time present */
  time_t mtime;                     /* last modified time         */
  ST_CHAR filename [MAX_FILE_NAME+1];
  } MVL_DIR_ENT;
```

### MVL_FDIR_RESP_INFO

This structure contains information for processing the **FileDirectory** response data.

```
typedef struct
  {
  ST_INT num_dir_ent;               /* number of directory entries */
  ST_BOOLEAN more_follows;          /* more dir entries follow     */
                                    /*   default: SD_FALSE         */
  MVL_DIR_ENT *dir_ent;             /* ptr to array of dir entries */
  } MVL_FDIR_RESP_INFO;
```

## *FileDirectory Functions*

### mvl_fdir

**Usage:** This function performs a synchronous FileDirectory request.

**Function Prototype:**

```
ST_RET mvl_fdir (MVL_NET_INFO *net_info,
                 ST_CHAR *filespec,
                 ST_CHAR *ca_filename,
                 MVL_REQ_PEND **req_out);
```

**Parameters:**

| | |
|---|---|
| `net_info` | Network connection information. |
| `filespec` | File specification for directory entries of interest (**`NULL`**-terminated string). |
| `ca_filename` | Name of file to continue after (**`NULL`**-terminated string). |
| `req_out` | See the description of **`req_out`** on page 183. |

**Response Data:** The response data is in the **`fdir`** parameter of the **`MVL_REQ_PEND`** structure. It may be referenced by the following statement (assuming **`&req_pend`** was passed as the **`req_out`** argument):

```
MVL_FDIR_RESP_INFO *resp_info = req_pend->u.fdir.resp_info;
```

**Return Value:**

| | | |
|---|---|---|
| `ST_RET` | `SD_SUCCESS` | If request sent and response received successfully, or error code. |

## mvla_fdir

**Usage:** This function performs an asynchronous FileDirectory request.

**Function Prototype:**

```
ST_RET mvla_fdir (MVL_NET_INFO *net_info,
                  ST_CHAR *filespec,
                  ST_CHAR *ca_filename,
                  MVL_REQ_PEND **req_out);
```

**Parameters:**

| | |
|---|---|
| `net_info` | Network connection information. |
| `filespec` | File specification for directory entries of interest (**`NULL`**-terminated string). |
| `ca_filename` | Name of file to continue after (**`NULL`**-terminated string). |
| `req_out` | See the description of **`req_out`** on page 183. |

**Response Data:** The response data is in the **`fdir`** parameter of the **`MVL_REQ_PEND`** structure. It may be referenced by the following statement (assuming **`&req_pend`** was passed as the **`req_out`** argument):

```
MVL_FDIR_RESP_INFO *resp_info = req_pend->u.fdir.resp_info;
```

**Return Value:** `ST_RET` `SD_SUCCESS` If request sent successfully, or error code.

**Note:** If the return value is **`SD_SUCCESS`**, you still need to wait for the response. See *Asynchronous Request Functions* on page 183 for information on how to wait.

# FileDelete Service

This service is used by a client to delete a file from the virtual filestore of a server.

## *FileDelete Functions*

### mvl_fdelete

**Usage:** This function performs a synchronous FileDelete request.

**Function Prototype:**

```
ST_RET mvl_fdelete (MVL_NET_INFO *net_info,
                    ST_CHAR *filename,
                    MVL_REQ_PEND **req_out);
```

**Parameters:**

| | |
|---|---|
| `net_info` | Network connection information. |
| `filename` | Name of file to delete (NULL-terminated string). |
| `req_out` | See the description of **`req_out`** on page 183. |

| | | | |
|---|---|---|---|
| **Return Value:** | `ST_RET` | `SD_SUCCESS` | If request sent and response received successfully, or error code. |

### mvla_fdelete

**Usage:** This function performs an asynchronous FileDelete request.

**Function Prototype:**

```
ST_RET mvla_fdelete (MVL_NET_INFO *net_info,
                     ST_CHAR *filename,
                     MVL_REQ_PEND **req_out);
```

**Parameters:**

| | |
|---|---|
| `net_info` | Network connection information. |
| `filename` | Name of file to delete (NULL-terminated string). |
| `req_out` | See the description of **`req_out`** on page 183. |

| | | | |
|---|---|---|---|
| **Return Value:** | `ST_RET` | `SD_SUCCESS` | If request sent successfully, or error code. |

**Note:** If the return value is **`SD_SUCCESS`**, you still need to wait for the response. See *Asynchronous Request Functions* on page 183 for information on how to wait.

# ObtainFile Service

This service is used by a Client application to cause the remote device to obtain a file from the local virtual file store.

## *ObtainFile Functions*

### mvl_obtfile

**Usage:** This function performs a synchronous ObtianFile request.

**Function Prototype:**

```
ST_RET mvl_obtfile (MVL_NET_INFO *net_info,
                    ST_CHAR *srcfilename,
                    ST_CHAR *destfilename,
                    MVL_REQ_PEND **req_out);
```

**Parameters:**

| | |
|---|---|
| `net_info` | Network connection information. |
| `srcfilename` | This is a NULL terminated ASCII string of the source file name on the local device. |
| `destfilename` | This is a NULL terminated ASCII string of the destination file name on the remote device. |
| `req_info` | ObtainFile request information. |

| | | | |
|---|---|---|---|
| **Return Value:** | `ST_RET` | `SD_SUCCESS` | If request sent and response received successfully. |
| | | `<>0` | Error code. |

## mvla_optfile

**Usage:** This function performs an asynchronous ObtianFile request.

**Function Prototype:**

```
ST_RET mvla_obtfile (MVL_NET_INFO *net_info,
                     ST_CHAR *srcfilename,
                     ST_CHAR *destfilename,
                     MVL_REQ_PEND **req_out);
```

**Parameters:**

| | |
|---|---|
| `net_info` | Network connection information. |
| `srcfilename` | This is a NULL terminated ASCII string of the source file name on the local device. |
| `destfilename` | This is a NULL terminated ASCII string of the destination file name on the remote device. |
| `req_info` | ObtainFile request information. |

| | | | |
|---|---|---|---|
| **Return Value:** | `ST_RET` | `SD_SUCCESS` | If request sent and response received successfully. |
| | | `<> 0` | Error code. |

# FileGet Service

FileGet is not a true MMS service. Rather it is a MMS-EASE *Lite* service that automatically generates MMS FileOpen, FileRead, and FileClose PDUs. The FileGet 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. If the file transfer is interupted or an error occurs during file transfer, no destination file will be created.

### *FileGet Data Structures*

```
typedef struct mvl_fget_req_info
  {
  ST_BOOLEAN fget_done;
  ST_INT fget_error;
  /* pointer to user's fget confirm function */
  ST_VOID (*fget_cnf_ptr)(struct mvl_fget_req_info *state);
  ST_CHAR srcfilename[MAX_FILE_NAME+1];
  ST_CHAR destfilename[MAX_FILE_NAME+1];
  ST_VOID *v;                /* For MVL user's use               */
  /* The rest of this structure is not normally accessed by the user.*/
  char tempfilename[L_tmpnam];
  FILE *fp;
  ST_INT32 frsmid;
  ST_UINT32 fsize;
  } MVL_FGET_REQ_INFO;
```

## *FileGet Functions*

## mvl_fget

**Usage:** This is a synchronous virtual machine function which 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.

**Function Prototype:**

```
ST_RET mvl_fget (MVL_NET_INFO  *net_info,
                 ST_CHAR *srcfilename,
                 ST_CHAR *destfilename,
                 MVL_FGET_REQ_INFO *fget_req_info);
```

**Parameters:**

| | |
|---|---|
| `net_info` | Network connection information. |
| `srcfilename` | This is a NULL terminated ASCII string of the source file name on the remote device. |
| `destfilename` | This is a NULL terminated ASCII string of the destination file name on the local device. |
| `fget_req_info` | This is a pointer to a pending service specific structure. Please refer to FileGet DataStructures for further information. |

| | | | |
|---|---|---|---|
| **Return Value:** | `ST_RET` | `SD_SUCCESS` | If request sent and response received successfully. |
| | | `<> 0` | Error code. |

## mvla_fget

**Usage:** This is an asynchronous virtual machine function which 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, and responses required by the MMS file operations or the operating system calls necessary to create the file locally.

**Function Prototype:**

```
ST_RET mvla_fget (MVL_NET_INFO *net_info,
                  ST_CHAR *srcfilename,
                  ST_CHAR *destfilename,
                  MVL_FGET_REQ_INFO *fget_req_info);
```

**Parameters:**

| | |
|---|---|
| `net_info` | Network connection information. |
| `srcfilename` | This is a NULL terminated ASCII string of the source file name on the remote device. |
| `destfilename` | This is a NULL terminated ASCII string of the destination file name on the local device. |
| `fget_req_info` | This is a pointer to a pending service specific structure. Please refer to FileGet Data Structures for further information. |

| | | | |
|---|---|---|---|
| **Return Value:** | `ST_RET` | `SD_SUCCESS` | If request sent and response received successfully |
| | | `<> 0` | Error code. |

# FileRename Service

This service is used by a Client application to Rename or move a file in a remote Virtual File Store.

## *FileRename Functions*

### mvl_frename

**Usage:** This function performs a synchronous FileRename request.

**Function Prototype:**

```
ST_RET mvl_frename (MVL_NET_INFO *net_info,
                    ST_CHAR *curfilename,
                    ST_CHAR *newfilename,
                    MVL_REQ_PEND **req_out);
```

**Parameters:**

| | |
|---|---|
| `net_info` | Network connection information. |
| `curfilename` | This is the NULL terminated ASCII string of the current file name in the remote virtual file store. |
| `newfilename` | This is the NULL terminated ASCII string of the new file name in the remote virtual file store. |
| `req_out` | See the description of **`req_out`** on page 183. |

| | | | |
|---|---|---|---|
| **Return Value:** | `ST_RET` | `SD_SUCCESS` | If request sent and response received successfully |
| | | `<> 0` | Error code. |

## mvla_frename

**Usage:** This function performs an asynchronous FileRename request.

**Function Prototype:**

```
ST_RET mvla_frename (MVL_NET_INFO *net_info,
                     ST_CHAR *curfilename,
                     ST_CHAR *newfilename,
                     MVL_REQ_PEND **req_out);
```

**Parameters:**

| | |
|---|---|
| `net_info` | Network connection information. |
| `curfilename` | This is the NULL terminated ASCII string of the current file name in the remote virtual file store. |
| `newfilename` | This is the `NULL` terminated ASCII string of the new file name in the remote virtual file store. |
| `req_out` | See the description of **`req_out`** on page 183. |

| | | | |
|---|---|---|---|
| **Return Value:** | `ST_RET` | `SD_SUCCESS` | If request sent successfully. |
| | | `<> 0` | Error code. |

# DefineNamedVariableList Service

This service is used by a Client application to request that a Server VMD create a NamedVariableList object. This allows access through a list of Named Variable objects, Unnamed Variable objects, or Scattered Access objects, or any combination.

## *DefineNamedVariableList Data Structures*

### DEFVLIST_REQ_INFO

See page 130 for more information.

## *DefineNamedVariableList Functions*

### mvl_defvlist

**Usage:** This function performs a synchronous DefineNamedVariableList request.

**Function Prototype:**

```
ST_RET mvl_defvlist (MVL_NET_INFO *net_info,
                     DEFVLIST_REQ_INFO *req_info,
                     MVL_REQ_PEND **req_out);
```

**Parameters:**

| | |
|---|---|
| `net_info` | Network connection information. |
| `req_info` | DefineNamedVariableList  request information. |
| `req_out` | See the description of **`req_out`** on page 183. |

| | | | |
|---|---|---|---|
| **Return Value:** | `ST_RET` | `SD_SUCCESS` | If request sent and response received successfully, or error code. |

## mvla_defvlist

**Usage:** This function performs an asynchronous DefineNamedVariableList request.

**Function Prototype:**

```
ST_RET mvla_defvlist (MVL_NET_INFO *net_info,
                      DEFVLIST_REQ_INFO *req_info,
                      MVL_REQ_PEND **req_out);
```

**Parameters:**

| | |
|---|---|
| `net_info` | Network connection information. |
| `req_info` | DefineNamedVariableList  request information. |
| `req_out` | See the description of **`req_out`** on page 183. |

**Return Value:** `ST_RET` `SD_SUCCESS` If request sent successfully, or error code.

**Note:** If the return value is **`SD_SUCCESS`**, you still need to wait for the response. See *Asynchronous Request Functions* on page 183 for information on how to wait.

# 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.

## *GetVariableAccessAttributes Data Structures*

### GETVAR_REQ_INFO

See page 126 for more information.

### GETVAR_RESP_INFO

See page 126 for more information.

## *GetVariableAccessAttributes Functions*

### mvl_getvar

**Usage:** This function performs a synchronous GetVariableAccessAttributes request.

**Function Prototype:**

```
ST_RET mvl_getvar (MVL_NET_INFO *net_info,
                   GETVAR_REQ_INFO *req_info,
                   MVL_REQ_PEND **req_out);
```

**Parameters:**

| | |
|---|---|
| `net_info` | Network connection information. |
| `req_info` | GetVariableAccessAttributes request information. |
| `req_out` | See the description of **`req_out`** on page 183. |

**Response Data:** The response data is in the `getvar` parameter of the **`MVL_REQ_PEND`** structure. It may be referenced by the following statement (assuming **`&req_pend`** was passed as the **`req_out`** argument):

```
GETVAR_RESP_INFO *resp_info = req_pend->u.getvar.resp_info;
```

| | | | |
|---|---|---|---|
| **Return Value:** | `ST_RET` | `SD_SUCCESS` | If request sent and response received successfully, or error code. |

## mvla_getvar

**Usage:** This function performs an asynchronous GetVariableAccessAttributes request.

**Function Prototype:**

```
ST_RET mvla_getvar (MVL_NET_INFO *net_info,
                    GETVAR_REQ_INFO *req_info,
                    MVL_REQ_PEND **req_out);
```

**Parameters:**

| | |
|---|---|
| `net_info` | Network connection information. |
| `req_info` | GetVariableAccessAttributes request information. |
| `req_out` | See the description of **`req_out`** on page 183. |

**Response Data:** The response data is in the **`getvar`** parameter of the **`MVL_REQ_PEND`** structure. It may be referenced by the following statement (assuming **`&req_pend`** was passed as the **`req_out`** argument):

```
GETVAR_RESP_INFO *resp_info = req_pend->u.getvar.resp_info;
```

**Return Value:** `ST_RET` `SD_SUCCESS` If request sent successfully, or error code.

**Note:** If the return value is **`SD_SUCCESS`**, you still need to wait for the response. See *Asynchronous Request Functions* on page 183 for information on how to wait.

# GetNamedVariableListAttributes Service

This service is used by a Client application to request that a Server VMD return the attributes of a NamedVariableList object defined at the VMD.

## *GetNamedVariableListAttributes Data Structures*

### GETVLIST_REQ_INFO

See page 139 for more information.

### GETVLIST_RESP_INFO

See page 139 for more information.

## *GetNamedVariableListAttributes Functions*

## mvl_getvlist

**Usage:** This function performs a synchronous GetNamedVariableListAttributes request.

**Function Prototype:**

```
ST_RET mvl_getvlist (MVL_NET_INFO *net_info,
                     GETVLIST_REQ_INFO *req_info,
                     MVL_REQ_PEND **req_out);
```

**Parameters:**

| | |
|---|---|
| `net_info` | Network connection information. |
| `req_info` | GetNamedVariableListAttributes request information. |
| `req_out` | See the description of **`req_out`** on page 183. |

**Response Data:** The response data is in the `getvlist` parameter of the **`MVL_REQ_PEND`** structure. It may be referenced by the following statement (assuming **`&req_pend`** was passed as the **`req_out`** argument):

```
GETVLIST_RESP_INFO *resp_info = req_pend->u.getvlist.resp_info;
```

| | | | |
|---|---|---|---|
| **Return Value:** | `ST_RET` | `SD_SUCCESS` | If request sent and response received successfully, or error code. |

## mvla_getvlist

**Usage:** This function performs an asynchronous GetNamedVariableListAttributes request.

**Function Prototype:**

```
ST_RET mvla_getvlist (MVL_NET_INFO *net_info,
                      GETVLIST_REQ_INFO *req_info,
                      MVL_REQ_PEND **req_out);
```

**Parameters:**

| | |
|---|---|
| `net_info` | Network connection information. |
| `req_info` | GetNamedVariableListAttributes request information. |
| `req_out` | See the description of **`req_out`** on page 183. |

**Response Data:** The response data is in the **`getvlist`** parameter of the **`MVL_REQ_PEND`** structure. It may be referenced by the following statement (assuming **`&req_pend`** was passed as the **`req_out`** argument):

```
GETVLIST_RESP_INFO *resp_info = req_pend->u.getvlist.resp_info;
```

**Return Value:** `ST_RET` `SD_SUCCESS` If request sent successfully. Otherwise, there will be an error code.

**Note:** If the return value is **`SD_SUCCESS`**, you still need to wait for the response. See *Asynchronous Request Functions* on page 183 for information on how to wait.

# GetDomainAttributes Service

This service is used to request that a Server return all of the attributes associated with a specific domain.

## *GetDomainAttributes Data Structures*

### GETDOM_REQ_INFO

See page 143 for more information.

### GETDOM_RESP_INFO

See page 143 for more information.

## *GetDomainAttributes Functions*

## mvl_getdom

**Usage:** This function performs a synchronous GetDomainAttributes request.

**Function Prototype:**

```
ST_RET mvl_status (MVL_NET_INFO *net_info,
                   GETDOM_REQ_INFO *req_info,
                   MVL_REQ_PEND **req_out);
```

**Parameters:**

| | |
|---|---|
| `net_info` | Network connection information. |
| `req_info` | GetDomainAttributes request information. |
| `req_out` | See the description of **`req_out`** on page 183. |

**Response Data:** The response data is in the **`getdom`** parameter of the **`MVL_REQ_PEND`** structure. It may be referenced by the following statement (assuming **`&req_pend`** was passed as the **`req_out`** argument):

```
GETDOM_RESP_INFO *resp_info = req_pend->u.getdom.resp_info;
```

**Return Value:**

| | | |
|---|---|---|
| `ST_RET` | `SD_SUCCESS` | If request sent and response received successfully, or error code. |

## mvla_getdom

**Usage:** This function performs an asynchronous GetDomainAttributes request.

**Function Prototype:**

```
ST_RET mvla_getdom (MVL_NET_INFO *net_info,
                    GETDOM_REQ_INFO *req_info,
                    MVL_REQ_PEND **req_out);
```

**Parameters:**

| | |
|---|---|
| `net_info` | Network connection information. |
| `req_info` | GetDomainAttributes request information. |
| `req_out` | See the description of **`req_out`** on page 183. |

**Response Data:** The response data is in the **`getdom`** parameter of the **`MVL_REQ_PEND`** structure. It may be referenced by the following statement (assuming **`&req_pend`** was passed as the **`req_out`** argument):

```
GETDOM_RESP_INFO *resp_info = req_pend->u.getdom.resp_info;
```

**Return Value:** `ST_RET` `SD_SUCCESS` If request sent successfully, or error code.

**Note:** If the return value is **`SD_SUCCESS`**, you still need to wait for the response. See *Asynchronous Request Functions* on page 183 for information on how to wait.

# DeleteNamedVariableList Service

This service is used by a Client application to request that a Server VMD delete one or more NamedVariablesList objects at a VMD. These must have a MMS Deletable attribute equal to true.

## *DeleteNamedVariableList Data Structures*

### DELVLIST_REQ_INFO

See page 134 for more information.

### DELVLIST_RESP_INFO

See page 135 for more information.

## *DeleteNamedVariableList Functions*

## mvl_delvlist

**Usage:** This function performs a synchronous DeleteNamedVariableList request.

**Function Prototype:**

```
ST_RET mvl_delvlist (MVL_NET_INFO  *net_info,
                     DELVLIST_REQ_INFO *req_info,
                     MVL_REQ_PEND **req_out);
```

**Parameters:**

| | |
|---|---|
| `net_info` | Network connection information. |
| `req_info` | DeleteNamedVariableList request information. |
| `req_out` | See the description of **`req_out`** on page 183. |

**Response Data:** The response data is in the `delvlist` parameter of the **`MVL_REQ_PEND`** structure. It may be referenced by the following statement (assuming **`&req_pend`** was passed as the **`req_out`** argument):

```
DELVLIST_RESP_INFO *resp_info = req_pend->u.delvlist.resp_info;
```

| | | | |
|---|---|---|---|
| **Return Value:** | `ST_RET` | `SD_SUCCESS` | If request sent and response received successfully, or error code. |

## mvla_delvlist

**Usage:** This function performs an asynchronous DeleteNamedVariableList request.

**Function Prototype:**

```
ST_RET mvla_delvlist (MVL_NET_INFO *net_info,
                      DELVLIST_REQ_INFO *req_info,
                      MVL_REQ_PEND **req_out);
```

**Parameters:**

| | |
|---|---|
| net_info | Network connection information. |
| req_info | DeleteNamedVariableList request information. |
| req_out | See the description of **req_out** on page 183. |

**Response Data:** The response data is in the **delvlist** parameter of the **MVL_REQ_PEND** structure. It may be referenced by the following statement (assuming **&req_pend** was passed as the **req_out** argument):

```
DELVLIST_RESP_INFO *resp_info = req_pend->u.delvlist.resp_info;
```

**Return Value:**

| | | |
|---|---|---|
| ST_RET | SD_SUCCESS | If request sent successfully, or error code. |

**Note:** If the return value is **SD_SUCCESS**, you still need to wait for the response. See *Asynchronous Request Functions* on page 183 for information on how to wait.

# 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.

## *InitializeJournal Data Structures*

### JINIT_REQ_INFO

See page 148 for more information.

### JINIT_RESP_INFO

See page 149 for more information.

## *InitializeJournal Functions*

## mvl_jinit

**Usage:** This function performs a synchronous InitializeJournal request.

**Function Prototype:**

```
ST_RET mvl_jinit (MVL_NET_INFO *net_info,
                  JINIT_REQ_INFO *req_info,
                  MVL_REQ_PEND **req_out);
```

**Parameters:**

| | |
|---|---|
| `net_info` | Network connection information. |
| `req_info` | InitializeJournal request information. |
| `req_out` | See the description of **`req_out`** on page 183. |

**Response Data:** The response data is in the **`jinit`** parameter of the **`MVL_REQ_PEND`** structure. It may be referenced by the following statement (assuming **`&req_pend`** was passed as the **`req_out`** argument):

```
JINIT_RESP_INFO *resp_info = req_pend->u.jinit.resp_info;
```

| **Return Value:** | | | |
|---|---|---|---|
| | `ST_RET` | `SD_SUCCESS` | If request sent and response received successfully, or error code. |

## mvla_jinit

**Usage:** This function performs an asynchronous InitializeJournal request.

**Function Prototype:**

```
ST_RET mvla_jinit (MVL_NET_INFO *net_info,
                   JINIT_REQ_INFO *req_info,
                   MVL_REQ_PEND **req_out);
```

**Parameters:**

| | |
|---|---|
| `net_info` | Network connection information. |
| `req_info` | InitializeJournal request information. |
| `req_out` | See the description of **`req_out`** on page 183. |

**Response Data:** The response data is in the **`jinit`** parameter of the **`MVL_REQ_PEND`** structure. It may be referenced by the following statement (assuming **`&req_pend`** was passed as the **`req_out`** argument):

```
JINIT_RESP_INFO *resp_info = req_pend->u.jinit.resp_info;
```

**Return Value:** `ST_RET` `SD_SUCCESS` If request sent successfully, or error code.

**Note:** If the return value is **`SD_SUCCESS`**, you still need to wait for the response. See *Asynchronous Request Functions* on page 183 for information on how to wait.

# 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.

## ReadJournal Data Structures

### JREAD_REQ_INFO

See page 152 for more information.

### MVL_JOURNAL_ENTRY

```
typedef struct
  {
  ST_INT entry_id_len;              /* Octet string ID, size 1-8  */
  ST_UCHAR *entry_id;

  APP_REF orig_app;
  MMS_BTOD occur_time;              /* occurrence time            */

  ST_INT16 entry_form_tag;          /* entry form tag             */
                                    /*  2 : data                  */
                                    /*  3 : annotation            */
  union
    {
    struct                          /* entry form is DATA         */
     {
      ST_BOOLEAN event_pres;        /* event present              */
      OBJECT_NAME evcon_name;       /* event condition name       */
      ST_INT16 cur_state;           /* current state              */
                                    /*  0 : disabled              */
                                    /*  1 : idle                  */
                                    /*  2 : active                */
      ST_BOOLEAN list_of_var_pres;  /* list of variables present  */
      ST_INT num_of_var;            /* number of variables        */
      VAR_INFO *list_of_var;        /* ptr to array               */
      } data;
    ST_CHAR *annotation;            /* pointer to annotation      */
    }ef;
  } MVL_JOURNAL_ENTRY;
```

### MVL_JREAD_RESP_INFO

```
typedef struct
  {
  ST_INT num_of_jou_entry;          /* number of journal entries  */
  ST_BOOLEAN more_follows;          /* default = false            */
  MVL_CLI_JOURNAL_ENTRY *jou_entry; /* ptr to array of entries    */
  } MVL_JREAD_RESP_INFO;
```

## *ReadJournal Functions*

### mvl_jread

**Usage:** This function performs a synchronous ReadJournal request.

**Function Prototype:**

```
ST_RET mvl_jread (MVL_NET_INFO *net_info,
                  JREAD_REQ_INFO *req_info,
                  MVL_REQ_PEND **req_out);
```

**Parameters:**

| | |
|---|---|
| `net_info` | Network connection information. |
| `req_info` | ReadJournal request information. |
| `req_out` | See the description of **`req_out`** on page 183. |

**Response Data:** The response data is in the **`jread`** parameter of the **`MVL_REQ_PEND`** structure. It may be referenced by the following statement (assuming **`&req_pend`** was passed as the **`req_out`** argument):

```
MVL_JREAD_RESP_INFO *resp_info = req_pend->u.jread.resp_info;
```

**Return Value:** `ST_RET` `SD_SUCCESS` If request sent and response received successfully, or error code.

## mvla_jread

**Usage:** This function performs an asynchronous ReadJournal request.

**Function Prototype:**

```
ST_RET mvla_jread (MVL_NET_INFO *net_info,
                   JREAD_REQ_INFO *req_info,
                   MVL_REQ_PEND **req_out);
```

**Parameters:**

| | |
|---|---|
| `net_info` | Network connection information. |
| `req_info` | ReadJournal request information. |
| `req_out` | See the description of **`req_out`** on page 183. |

**Response Data:** The response data is in the **`jread`** parameter of the **`MVL_REQ_PEND`** structure. It may be referenced by the following statement (assuming **`&req_pend`** was passed as the **`req_out`** argument):

```
MVL_JREAD_RESP_INFO *resp_info = req_pend->u.jread.resp_info;
```

**Return Value:** `ST_RET` `SD_SUCCESS` If request sent successfully, or error code.

**Note:** If the return value is **`SD_SUCCESS`**, you still need to wait for the response. See *Asynchronous Request Functions* on page 183 for information on how to wait.

# ReportJournalStatus Service

This service is used to determine the number of entries in a Journal object.

## *ReportJournalStatus Data Structures*

### JSTAT_REQ_INFO

See page 158 for more information.

### JSTAT_RESP_INFO

See page 158 for more information.

## *ReadJournalStatus Functions*

## mvl_jstat

**Usage:** This function performs a synchronous ReportJournalStatus request.

**Function Prototype:**

```
ST_RET mvl_jstat (MVL_NET_INFO *net_info,
                  JSTAT_REQ_INFO *req_info,
                  MVL_REQ_PEND **req_out);
```

**Parameters:**

| | |
|---|---|
| `net_info` | Network connection information. |
| `req_info` | ReportJournalStatus request information. |
| `req_out` | See the description of **`req_out`** on page 183. |

**Response Data:** The response data is in the **`jstat`** parameter of the **`MVL_REQ_PEND`** structure. It may be referenced by the following statement (assuming **`&req_pend`** was passed as the **`req_out`** argument):

```
JSTAT_RESP_INFO *resp_info = req_pend->u.jstat.resp_info;
```

**Return Value:** `ST_RET` `SD_SUCCESS` If request sent and response received successfully, or error code.

## mvla_jstat

**Usage:** This function performs an asynchronous ReportJournalStatus request.

**Function Prototype:**

```
ST_RET mvla_jstat (MVL_NET_INFO *net_info,
                   JSTAT_REQ_INFO *req_info,
                   MVL_REQ_PEND **req_out);
```

**Parameters:**

| | |
|---|---|
| `net_info` | Network connection information. |
| `req_info` | ReportJournalStatus request information. |
| `req_out` | See the description of **`req_out`** on page 183. |

**Response Data:** The response data is in the **`jstat`** parameter of the **`MVL_REQ_PEND`** structure. It may be referenced by the following statement (assuming **`&req_pend`** was passed as the **`req_out`** argument):

```
JSTAT_RESP_INFO *resp_info = req_pend->u.jstat.resp_info;
```

**Return Value:**

| | | |
|---|---|---|
| `ST_RET` | `SD_SUCCESS` | If request sent successfully, or error code. |

**Note:** If the return value is **`SD_SUCCESS`**, you still need to wait for the response. See *Asynchronous Request Functions* on page 183 for information on how to wait.

Chapter 7

# Using MVL UCA Support

To provide UCA support, MVLU makes use of standard MVL features such as the Manufactured Object Handlers and Indication Handlers.

## Read/Write Indication Functions

MMS Object Foundry generates code to allow the MVLU support library to invoke user provided functions to implement the MMS Read and Write services. This code makes use of the concept of Read/Write indication handing functions for all primitive data elements of a UCA type. Please note that these indication functions are NOT associated with a particular variable, but rather with a type. This means that if there is more than one variable of a type it is necessary to use the base Variable Association to determine which variable is being accessed.

### Read Indication Functions

The Read Indication functions have prototypes of the following form:

```
ST_VOID u_xxx_yyy_zzz_rd_ind_fun (MVLU_RD_VA_CTRL *mvluRdVaCtrl)
```

where **`xxx_yyy_zzz`** is created by MMS Object Foundry and is based on the UCA name of the primitive level object. For instance, for the UCA Device Identity (DI) object, the following Read Indication Function names are used:

```
u_di_name_rd_ind_fun
u_di_own_rd_ind_fun
u_di_vndid_devmdls_rd_ind_fun
u_di_vndid_sftrev_rd_ind_fun
u_di_commid_pro_rd_ind_fun
u_di_class_rd_ind_fun
u_di_loc_rd_ind_fun
u_di_vndid_sernum_rd_ind_fun
u_di_commid_commadr_rd_ind_fun
u_di_commid_med_rd_ind_fun
u_di_d_rd_ind_fun
u_di_vndid_vnd_rd_ind_fun
u_di_vndid_hwrev_rd_ind_fun
u_di_commid_commrev_rd_ind_fun
u_di_commid_mac_rd_ind_fun
```

The **MVLU_RD_VA_CTRL** data structure is passed into the read indication handler functions. It is used to allow the user application to handle each primitive data element read separately and asynchronously. MVLU keeps track of the number of **MVLU_RD_VA_CTRL** outstanding for a READ or WRITE indication and sends the MMS response when all have been handled. See the figure on page 241.

```
typedef struct mvlu_rd_va_ctrl
 {
 MVL_IND_PEND *indCtrl;
 MVLAS_RD_VA_CTRL *rdVaCtrl;
 ST_CHAR *primData;
 ST_RTREF primRef;
 ST_UINT prim_num;
 ST_UINT prim_offset_base;
} MVLU_RD_VA_CTRL;
```

where:

`indCtrl` This is a pointer to the MVL indication control structure for the MMS indication. This structure contains two user-controlled fields that can be used to manage indication wide user information.

`rdVaCtrl` This is a pointer to the MVL Variable Association structure for the MMS variable being accessed. In this data structure are several elements that are useful in processing the Read Indication such as a reference to the Base VA from which this VA was derived. This Base VA is the high level configured VA and is used to distinguish between variables of the same type.

Note that the **rdVaCtrl** structure is NOT unique to this particular Read Indication Function (i.e., single MMS Variable Specification can result in many primitive indication functions in the case of a structure type variable).

`primData` This is a pointer to data buffer for the primitive variable data for a Read Indication. This is where the data to be returned is to be placed. Note that there is a single data buffer for each MMS variable (MVL Variable Association) and the **primData** points somewhere into this buffer. The VA data buffer is normally allocated dynamically. Please refer to the *UCA Buffer Management* on page 251 for more information.

`primRef` This is the primitive element reference, which is controlled by the developer. MMS Object Foundry makes use of a MMS Object Foundry generated define to initialize the reference element for each primitive element. The developer can modify this define using a Template input file and then can use this reference to aid processing of the indication. This can be especially useful when the developer chooses to use a single Read Indication Function to support access to multiple primitive elements.

`prim_num` Index to data (0, 1, 2, 3, etc.). Unique for each primitive data element. Starts at 0 for the first primitive data element in the base variable.

`prim_offset_base` Memory offset (in bytes) of this primitive data element from start of base variable.

When the read response data has been put into the buffer selected by the **primData** element, the user application must call the MVLU function **mvlu_rd_prim_done** so that MVLU can send the read response. Note that this can be either within the Read Indication Function or asynchronously some time later.

```
ST_VOID mvlu_rd_prim_done (MVLU_RD_VA_CTRL *mvluRdVaCtrl, ST_RET rc);
```

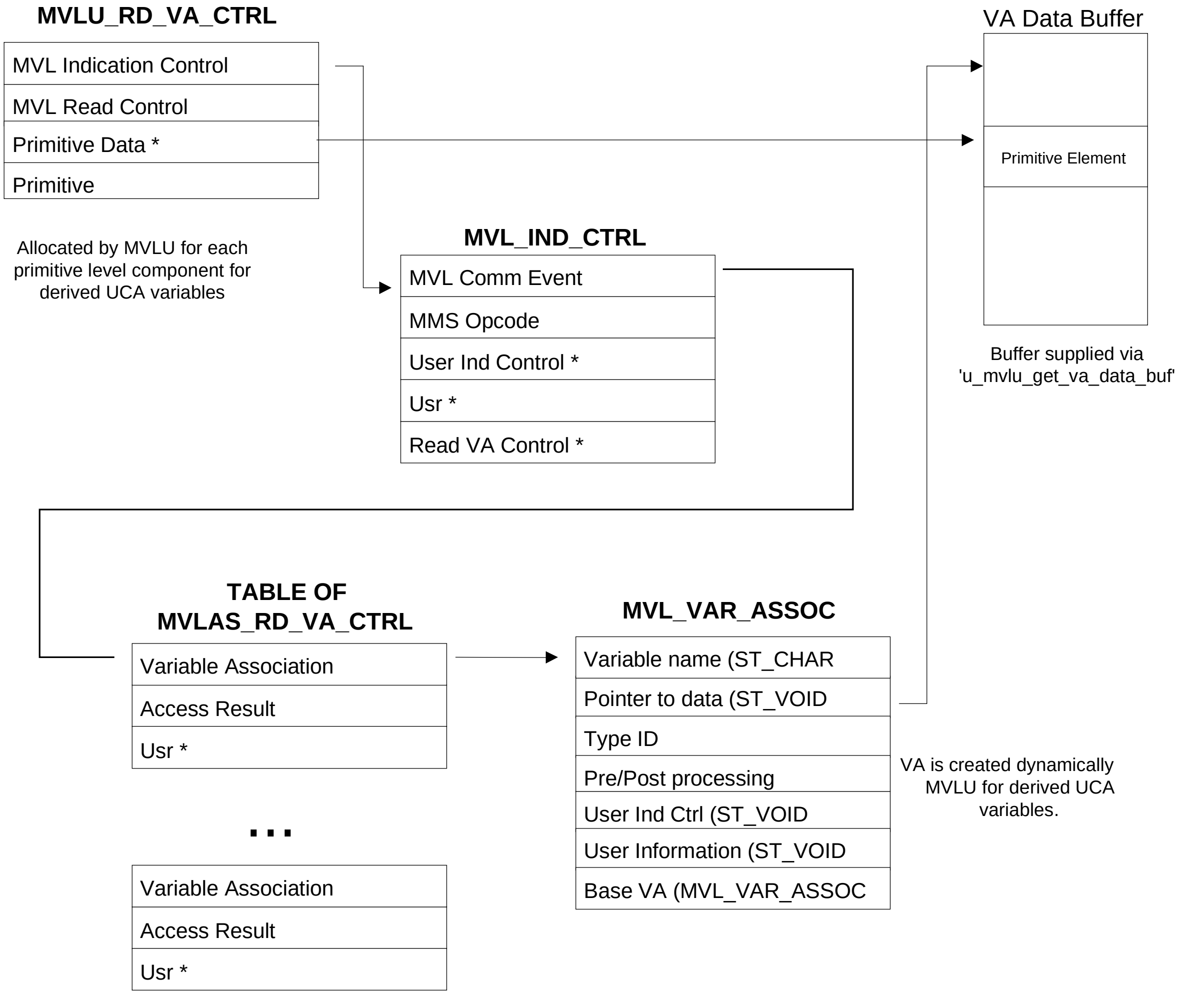


Figure 11: MVLU Read Control

# Write Indication Functions

The Write Indication Function concepts are the same as those used in the Read Indication Functions. The differences are described below:

The Write Indication functions have prototypes of the following form:

```
ST_VOID u_xxx_yyy_zzz_wr_ind_fun (MVLU_WR_VA_CTRL *mvluWrVaCtrl)
```

The **MVLU_WR_VA_CTRL** structure is declared as shown below and provides all required context information to allow the Write Indication Function to process the primitive data effectively.

```
typedef struct mvlu_wr_va_ctrl
  {
  MVL_IND_PEND *indCtrl;
  MVLAS_WR_VA_CTRL *wrVaCtrl;
  ST_CHAR *primData;
  ST_RTREF primRef;
  ST_UINT prim_num;
  ST_UINT prim_offset_base;
  } MVLU_WR_VA_CTRL;
```

When the write data located in the buffer selected by the **primData** element has been processed by the application, the user application must call the MVLU function **mvlu_wr_prim_done** so that MVLU can send the write response. Note that this can be either within the Write Indication Function or asynchronously some time later.

```
ST_VOID mvlu_wr_prim_done (MVLU_WR_VA_CTRL *mvluWrVaCtrl, ST_RET rc);
```

# Dynamic Type Creation for UCA and IEC-61850

***NOTE:*** *This section uses the term "Leaf Access Parameters" (LAP) to refer to "leaf functions and references" used in the UCA and IEC-61850 Object modeling.*

Dynamic type creation for UCA and IEC-61850 devices requires extra code to set the "Leaf Acccess Parameters" (LAP) after the type is created. The functions in this section simplify the process by allowing the following:

- Programmatic access to Leaf Access Parameters (i.e., find leaf nodes by name, set leaf access parameters).
- Runtime loading of LAP information from XML file, or from any user source.

### *Dynamically Creating IEC-61850 Types from Input Obtained from the SCL File*

When using SCL to configure an IEC-61850 Server application, some of the options of the SCL language are not useful. Therefore, the following restrictions are placed on the SCL file used to configure the "61850 Server:"

- It must contain at least one ***IED*** section and, within that, at least one ***AccessPoint*** section. There must be one ***IED*** element whose "name" attribute matches the **iedName** argument passed to **scl_parse**. Within that *IED* element, there must be an ***AccessPoint*** element whose "name" attribute matches the **accessPointName** argument passed to **scl_parse**.
  *NOTE:* In the scl_srvr sample application , the ***IEDName*** and ***AccessPointName*** parameters are extracted from **startup.cfg**.
- The ***AccessPoint*** section may contain only the ***Server*** element. The SCL language allows for one ***Server*** element or multiple ***LN*** elements. The ***LN*** element is not well defined, and it may not contain all the information needed to configure a IEC-61850 Server, so it is not allowed.

**NOTE**: Only the necessary information is extracted from the SCL file. The SCL parser skips over large sections of the file.

The following functions allow dynamic creation of IEC-61850 types, Logical Devices, Logical Nodes and Report Control Blocks from input obtained from the SCL file (the SCL file format is defined by IEC-61850-6. The typical user does not need to know anything about the **SCL_INFO** structure used to store information extracted from the SCL file, but if needed, it is defined in **scl.h**.

## scl_parse

**Usage:** Completely parses an SCL file and stores all information extracted for a single **AccessPoint** element within a single **IED** element into a single structure.

**Function Prototype:**

```
ST_RET scl_parse (ST_CHAR *xmlFileName,
                  ST_CHAR *iedName,
                  ST_CHAR *accessPointName,
                  SCL_INFO *sclInfo);
```

**Parameters:**

| | |
|---|---|
| xmlFileName | Name of SCL file to parse (e.g. scl.xml). |
| iedName | Extract information from the SCL file **ONLY** from the **IED** element whose "name" attribute matches this name. All other **IED** elements are ignored. |
| accessPointName | Extract information from the SCL file **ONLY** from the **AccessPoint** element whose "name" attribute matches this name. All other **AccessPoint** elements are ignored. **The AccessPoint element must be contained within the IED element.** |
| sclInfo | Pointer to structure in which to store all information extracted from the SCL file. This must point to a previously allocated structure (or possibly a local or global variable). This function completely initializes the structure, so there is no need to calloc or memset the structure before calling this function. |

| | | | |
|---|---|---|---|
| **Return Value:** | ST_RET | SD_SUCCESS | No Error |
| | | != SD_SUCCESS | Error |

## scl2_datatype_create_all

**Usage:** Creates MMS Data types for all Logical Node Types (LNodeType) defined in the SCL file. This function requires a user buffer for constructing TDL strings.

***NOTE:*** *The function* ***`scl_parse`*** *must be called first to read the SCL file and initialize the* ***`SCL_INFO`*** *structure passed to this function.*

**Function Prototype:**

```
ST_RET scl2_datatype_create_all(SCL_INFO *sclInfo,
                                ST_CHAR *tdlbuf,
                                size_t tdlbuflen);
```

**Parameters:**

`sclInfo` Pointer to structure containing all information extracted from the SCL file (filled in by **`scl_parse`**).

`tdlbuf` Buffer in which to construct TDL for each Logical Node Type (LNodeType) found in the SCL file. Reused for each LNodeType in list. . The buffer must be large enough to construct the TDL for every LNodeType defined in the SCL file.

`tdlbufname` Length of **`tdlbuf`** buffer in bytes.

| | | | |
|---|---|---|---|
| **Return Value:** | `ST_RET` | `SD_SUCCESS` | No Error |
| | | `!= SD_SUCCESS` | Error |

## scl2_ld_create_all

**Usage:** Creates all Logical Devices from information extracted from the SCL file. This includes creating the Logical Device (MMS Domain), and within the Logical Device: all Logical Nodes (MMS variables), all DataSets (MMS NamedVariableLists), and all ReportControlBlocks.

***NOTE:*** *The functions **`scl_parse`** and **`scl2_datatype_create_all`** must be called first to read the SCL file and initialize the **`SCL_INFO`** structure passed to this function, and to create all MMS Data Types needed by the Logical Device.*

**Function Prototype:**

```
ST_RET scl2_ld_create_all (SCL_INFO *sclInfo,
                           ST_UINT reportScanRate);
```

**Parameters:**

`sclInfo` Pointer to structure containing all information extracted from the SCL file (filled in by **`scl_parse`**).

`reportScanRate` Report Scan Rate (in milliseconds). If this value is not 0 (zero) and Report Control Blocks (RCB) are configured in the SCL file, a Scan Control object with this scan rate is automatically created for each RCB by calling **`mvlu_rpt_create_scan_ctrl2`**. If this value is 0 (zero), Scan Control objects are not created, and another method must be used to detect data changes for Reports.

| | | | |
|---|---|---|---|
| **Return Value:** | `ST_RET` | `SD_SUCCESS` | No Error |
| | | `!= SD_SUCCESS` | Error |

The following support functions are required by **`scl2_ld_create_all`** when creating Report Control Blocks (RCB), to allow dynamic initialization of Leaf Access Parameters (LAP). The function **`mvlu_set_leaf_param_name`** may also be called directly by the user application to set Leaf Access Parameters for any leaf.

## mvlu_set_leaf_param_name

**Usage:** This function is used to set the Leaf parameter name. Notice that the function names are passed as strings (not function pointers). This function looks up the function by name and sets the function pointer in the type definition. There is no need for the user to convert the function name to a function pointer.

***NOTE:*** *This function is only available if* ***`MVLU_LEAF_FUN_LOOKUP_ENABLE`*** *is defined (preferably in* ***glbopt.h****).*

**Function Prototype:**

```
ST_RET mvlu_set_leaf_param_name (ST_INT setFlags,
                                 ST_CHAR *leafName,
                                 ST_CHAR *rdIndFunName,
                                 ST_CHAR *wrIndFunName,
                                 ST_CHAR *refString);
```

**Example:**

```
rc = mvlu_set_leaf_param_name (MVLU_SET_ALL, “DI$VndID$SerNum”,
               “u_string_offset_rd_ind_fun”, “u_no_write_allowed”, “0”);
```

**Parameters:**

`setFlags` Indicates one or more of the following defines:

```
#define MVLU_SET_RD_FUN 0x01
#define MVLU_SET_WR_FUN 0x02
#define MVLU_SET_REF    0x04
#define MVLU_SET_ALL (MVLU_SET_RD_FUN | MVLU_SET_WR_FUN |
MVLU_SET_REF)
```

`*leafName` This is the name of the leaf.

`*rdIndFunName` This is the read indication function name.

`*wrIndFunName` This is the write indication function name.

`*refString` This is the string to convert to the “reference.”

| | | | |
|---|---|---|---|
| **Return Value:** | `ST_RET` | `SD_SUCCESS` | No Error |
| | | `!= SD_SUCCESS` | Error |

## u_mvlu_resolve_leaf_ref

**Usage:** The **`refString`** argument to **`mvlu_set_leaf_param_name`** must be converted to **`ST_RTREF`** to be stored as the **`reference`** in the type definition. Because users may wish to store almost anything in the **`reference`**, a user callback function, **`u_mvlu_resolve_leaf_ref`**, is called to convert the string into a **`ST_RTREF`** value. In the simplest case, the string may contain an integer value, in which case this function may simply convert the string to an integer using **`sscanf`** or **`atoi`** and then cast the value to **`ST_RTREF`**. An example of **`u_mvlu_resolve_leaf_ref`** is provided in **uca_srvr.c**.

**Function Prototype:**

```
ST_RET u_mvlu_resolve_leaf_ref (ST_CHAR *leafName,
                                ST_INT *setFlagsIo,
                                ST_CHAR *refString,
                                ST_RTREF *refOut);
```

**Parameters:**

`*leafName` This is the same leaf name passed to **`mvlu_set_leaf_param_name`**.

`*setFlagsIo` Indicates one or more of the following defines:

```
#define MVLU_SET_RD_FUN 0x01
#define MVLU_SET_WR_FUN 0x02
#define MVLU_SET_REF    0x04
#define MVLU_SET_ALL (MVLU_SET_RD_FUN | MVLU_SET_WR_FUN |
MVLU_SET_REF)
```

`*refString` This is same string to convert to the “reference.”.

`*refOut` This is output value converted from **`refString`**.

| **Return Value:** | `ST_RET` | `SD_SUCCESS` | No Error |
|---|---|---|---|
| | | `!= SD_SUCCESS` | Error |

## mvlu_load_xml_leaf_file

**Usage:** This function loads text based LAP information from an XML file and calls **mvlu_set_leaf_param_name** to set the Leaf Access Parameters for each leaf. Below is a very simple example of an XML input file:

```
<Leafmap>
  <Leaf Name="DI$Name" RdIndFun="rdString" WrIndFun="noWrite" Ref="42"/>
  <Leaf Name="DI$Class" RdIndFun="rdString" WrIndFun="noWrite" Ref="43"/>
</Leafmap>
```

The function syntax is very simple. The only argument is the name of the XML file to use as input.

**NOTE:** This function is only available if **MVLU_LEAF_FUN_LOOKUP_ENABLE** is defined (preferably in **glbopt.h**).

**Function Prototype:** `ST_RET mvlu_load_xml_leaf_file (ST_CHAR *fileName);`

**Parameters:**

| | |
|---|---|
| *leafName | This is the leaf name. |

| | | | |
|---|---|---|---|
| **Return Value:** | ST_RET | SD_SUCCESS | No Error |
| | | != SD_SUCCESS | Error |

## mvlu_set_leaf_param

**Usage:** This lower level function may also be used to set one or more Leaf Access Parameters if function indices (**`rdIndFunIndex`**, **`wrIndFunIndex`**) can be determined by some other method.

**Function Prototype:**

```
ST_RET mvlu_set_leaf_param (ST_INT setFlags,
                            ST_CHAR *leafName,
                            ST_RTINT rdIndFunIndex,
                            ST_RTINT wrIndFunIndex,
                            ST_RTREF ref);
```

**Example:**

```
   rc = mvlu_set_leaf_param (MVLU_SET_ALL, “DI$VndID$SerNum”,
                   (ST_RTINT)42,(ST_RTINT)43,(ST_RTREF)44);
```

**Parameters:**

`setFlags` Indicates one or more of the following defines:

```
#define MVLU_SET_RD_FUN 0x01
#define MVLU_SET_WR_FUN 0x02
#define MVLU_SET_REF    0x04
#define MVLU_SET_ALL (MVLU_SET_RD_FUN | MVLU_SET_WR_FUN |
MVLU_SET_REF)
```

`*leafName` This is the leaf name.

`rdIndFunIndex` This is the read indication function index.

`wrIndFunIndex` This is the write indication function index.

`ref` This is “reference.”

| Return Value: | ST_RET | SD_SUCCESS | No Error |
|---|---|---|---|
| | | != SD_SUCCESS | Error |

- **Adding Named MVL Types**

The function **`mvlu_add_rt_type_x`** has been added. This function is an extension to the existing function **`mvlu_add_rt_type`**, and allows the user to set the type name field. Note that the type name must be unique.

- **Type Name to Type ID Lookup**

This function can be used to find a Type by name, and returns the type id, or –1 in case of error. Note that the type must be created by Foundry OR added using **`mvlu_add_rt_type_x`**.

```
   ST_INT mvl_typename_to_typeid (ST_CHAR *typename);
```

Example:

```
di_typeid = mvl_typename_to_typeid (“DI”);
if (di_typeid < 0)
  Failure;
```

The inverse function is also available, and may be useful for logging and diagnostics.

```
ST_CHAR *mvl_typeid_to_typename (ST_INT type_id);
```

- **Finding A Runtime Type Element By Type ID And Leaf Name**

  This function is used to locate a particular leaf node within the selected type. This is useful if the developer wants to directly access the leaf's Runtime Type element.

```
RUNTIME_TYPE *mvlu_find_rt_leaf (ST_INT type_id,
                              ST_CHAR *leafName);
```

Example:

```
rt = mvlu_find_rt_leaf (di_type_id, “VndID$SerNum”);
if (rt == NULL)
  Failure;
```

# Array Handling

MVLU can handle single dimensional arrays of any support data type, including primitive and complex types. These are handled with multiple calls to the primitive leaf functions (one call per array element); Read/Write functions have access to the element index. As with UCA data structures, alternate access is handled transparently with the Read/Write function called for selected array elements only.

Foundry generates index use code for stub starter Read/Write functions in the following format:

```
{
...
ST_RTINT curr_index;
  curr_index = mvluRdVaCtrl->rdVaCtrl->va->arrCtrl.curr_index;
...
}
```

Please see the MVL UCA sample application the directory **\mmslite\mvl\usr\uca_srvr**.

# Template File (Obsolete)

***This file should not be used when developing new applications. Use the "LAP XML Input File" instead. It is supported only for backward compatibility with previously developed applications.***

When UCA processing is enabled, MMS Object Foundry can take a Template File as a source of Read/Write Indication Function and Reference information. When MMS Object Foundry needs to provide a Read or Write Indication Function, it searches the Template File in the following manner to resolve this required function.

1. MMS Object Foundry searches for a define for the function name. If it is found, the value is used as the name of the function to be used for that primitive element.
2. Assuming the define for the function name is not found, MMS Object Foundry then searches for an extern declaration for the function. If this is found, MMS Object Foundry assumes that the function will be supplied in a separate C module.
3. Assuming no define or extern is found for the function name, MMS Object Foundry then searches the Template for the function itself. If found, the code for the function is copied into the output C file.
4. If the function name is not found in any of the above forms, MMS Object Foundry generates a starter function and places it into the output C file and in the **lefttodo.txt** output file. These starter functions will allow the application to be compiled, linked and run, with simulated data supplied. Note that the starter functions are intended to be edited to provide real functionality and then moved into the input Template File.

In similar fashion, when MMS Object Foundry needs to provide a Reference define, it searches the Template File in the following manner to resolve this required define statement.

1. MMS Object Foundry searches for the define in the Template File. If it is found, it is extracted and placed in the output H file.
2. If the reference define is not found in the Template File, MMS Object Foundry generates a starter define and places it into the output H file and in the **lefttodo.txt** output file. These starter defines will allow the application to be compiled, linked and run. Note that the starter reference defines are intended to be edited to provide real functionality and then moved into the input Template File.

# VA Processing Functions and UCA Variables

The standard MVL pre/post processing functions for Read/Write/Info Report services on Variable Associations are supported for UCA variables. However, they are not normally required. To use this feature, the Read/Write Indication Function must set the **`proc`** element of the derived Variable Association to select the desired processing functions.

# Combining UCA and Non-UCA Variables

MVLU fully supports the standard non-UCA variables, with a few minor modifications. Since MVLU installs handlers for READ and WRITE indications, applications that also have non-UCA variables must consider the following:

1. By default, responses will be sent immediately for non-UCA variables per the normal UCA service mechanisms. Note that all pre/post processing functions will be called just as though MVLU was not installed.
2. To make use of the asynchronous response capability of MVL, the developer must set the function pointers below to a user defined function. If this is done, the application must call **`mvlu_rd_prim_done`** when the va->data is ready (READ), or **`mvlu_wr_prim_done`** when the va->data has been dealt with appropriately.

```
/* Function pointers for non-UCA variable handling */

ST_VOID(*mvluAsyncRdIndFun)(struct mvlu_rd_va_ctrl *mvluRdVaCtrl);

ST_VOID(*mvluAsyncWrIndFun)(struct mvlu_wr_va_ctrl *mvluWrVaCtrl);
```

# UCA Buffer Management

When a Read/Write indication is received for a UCA variable, MVLU allocates a data buffer that is appropriately sized for the MMS variable and then calls primitive indication handler functions to handle the individual data elements. The VA data buffer is allocated and freed via these static functions.

```
static ST_VOID mvluDefGetVaDataBufFun (ST_INT service,
                                       MVL_VAR_ASSOC *va,
                                       ST_INT size);

static ST_VOID mvluDefFreeVaDataBufFun (ST_INT service,
                                        MVL_VAR_ASSOC *va);
```

These functions check the **`use_static_data`** flag in the **`MVL_VAR_ASSOC`** structure for the base variable. If **`use_static_data`** is **`SD_FALSE`**, **`chk_calloc`** and **`chk_free`** are called to allocate and free the VA data buffer. If **`use_static_data`** is **`SD_TRUE`**, MVLU will assume that the configured (base) VA data element points to a user selected data buffer of the size of the base VA. MVLU will then set the derived VA data pointer to that base data pointer plus the calculated offset of the data element. This is useful when the application actually has the UCA object in memory.

When a MMS read or Write indication is received, functions selected by the function pointers shown below are invoked in order to allow the user to prepare for handling the indication. A typical use would be to use the **`indCtrl->usr_ind_ctrl`** to assist the buffer control subsystem to work effectively.

```
/* These function pointers are invoked to allow the user    */
/* application    to prepare for handling the indication.   */
extern ST_RET (*u_mvl_rd_ind_start) (MVL_IND_PEND *indCtrl);
extern ST_RET (*u_mvl_wr_ind_start) (MVL_IND_PEND *indCtrl);
```

Please note that this feature and other buffer management issues are advanced options. The default MVLU buffer management will work well for most applications.

# MVL UCA Report Handling

MVLU contains a set of functions and data structures that are useful in handling the UCA report control blocks and associated data sets. The MVL report handling system is based on the data structure **MVLU_RPT_CTRL**, which allows the application programmer to generate UCA reports easily. Please see the sample server source **uca_srvr.c** for an example of the use of these functions.

## *The MVLU Report Control Element*

The data structures below are used by MVLU to represent the BASRCB report control object.

```
typedef struct
  {
/* Each connection get it's own view of this data                     */
  ST_BOOLEAN RptEna;
  ST_CHAR    RptID[66];
  ST_CHAR   *DatSetNa[66];      /* Read only, get from Data Set NVL   */
  struct     /* BVstring */
    {
    ST_INT16 len_1;
    ST_UCHAR data_1[2];
    } OptFlds;
  ST_UINT32 BufTim;
  ST_UINT16 Trgs;
  ST_UINT8  SeqNum;
  ST_UCHAR  TrgOps[1];
  ST_UINT32 RBEPd;
  ST_UINT32 IntgPd;
  ST_UINT32 ConfRev;         /* Used only for IEC-61850 BRCB/URCB.    */
  ST_INT    EntryID;         /* Used only for IEC-61850 BRCB.         */
  } MVLU_BASRCB;

typedef struct mvlu_rpt_ctrl
  {
  DBL_LNK l;                  /* Internal use   */

/* Active clients */
  ST_INT num_rpt_clients;
  MVLU_RPT_CLIENT *rpt_client_list;

/* basrcb data for passive read clients */
  MVLU_BASRCB common_basrcb;
  ST_CHAR *basrcb_name;

/* Used in read/write indication functions in finding the report ctrl */
  RUNTIME_TYPE *rcbRtHead;
  MVL_VAR_ASSOC *base_va;

/* Used to support different different report schemes             */
  ST_INT rcb_type;      /* RCB_TYPE_UCA, RCB_TYPE_IEC_BRCB, etc.  */
/* Action to be taken if var changes twice before buftim expires  */
  ST_INT buftim_action; /*MVLU_RPT_BUFTIM_REPLACE/SEND_NOW        */

/* Report Data Fields, used as data source when report is sent */
  ST_UINT8 *inclusion_data;

/* The information below is used internally by MVLU               */
  MVL_VAR_ASSOC rptID_va;
  MVL_VAR_ASSOC optFlds_va;
  MVL_VAR_ASSOC sqNum_va;
  MVL_VAR_ASSOC inclusion_va;
  MVL_VAR_ASSOC *reasons_va;
  MVL_NVLIST_CTRL *dsNvl;     /* The base dataSet for the report  */
  MVL_NVLIST_CTRL rptNvl;     /* The NVL used to send the InfoRpt */
  RUNTIME_TYPE incRt;         /* Used in building the inclusion_va*/
  ST_INT maxNumRptVars;       /* Max vars used in report.         */
  MVLU_RPT_TYPEIDS rpt_typeids;   /* Types needed for reports.    */
  MVL61850_BRCB_CTRL brcbCtrl;    /* Used only for 61850 BRCB.    */
  } MVLU_RPT_CTRL;
```

## UCA Reporting Setup Sequence

The following steps should be followed to configure and enable UCA Reporting.

1. **Include the types required for UCA reporting.** To do this, include the following lines in your project’s ODF file, which can be done by including the following line in the ODF file:

   ```
   ":CI", "gentypes.odf"
   ":CI", "rpt.odf"
   ```

2. **For each UCA report, create the dataset NVL.** To do this you must identify the variable(s) within the brick that contains the variables to be reported. For instance, the GLOBE brick contains the report control element **`GLOBE$RP$brcbST`**, which has a dataset that includes all ST variables from the logical device. For a logical device with a single PBRO brick, this means that the dataset includes variables in the **`PBRO$ST`** data structure. These variables are:

   ```
      PBRO$ST$Out
      PBRO$ST$Tar
      PBRO$ST$FctDS
      PBRO$ST$AuxIn1
      PBRO$ST$PuGrp
   ```

   The easiest way to create this dataset is to use the following function:

   ```
   MVL_NVLIST_CTRL *mvlu_derive_rpt_ds (ST_CHAR *domName,
                                        ST_CHAR *nvlName,
                                        ST_INT numNodes,
                                        ST_CHAR **nodeNames);
   ```

   An example of the use of this function is as follows:

   ```
      ST_CHAR *nodeNames[1];
      nodeNames[0] = "PBRO$ST";
      nvl = mvlu_derive_rpt_ds ("pbroDev", "globeStRptDs", 1,
      nodeNames);
   ```

   See page 257 for more information on this function.

Alternatively, you can use the function **`mvlu_rpt_nvl_add`** to create the report dataset NVL. This function takes the desired NVL name along with a table of MMS variable names to be included in the NVL.

```
MVL_NVLIST_CTRL * mvlu_rpt_nvl_add  (OBJECT_NAME *nvl_obj,
                                    ST_INT num_var,
                                    OBJECT_NAME *var_obj);
```

An example of the use of this function is:

```
   OBJECT_NAME nvlObjName;
   OBJECT_NAME varObjNames[5];
   ST_INT i;

     nvlObjName.object_tag = DOM_SPEC;
     nvlObjName.domain_id = "pbroDev";
     nvlObjName.obj_name.item_id = "LogDev$ST";

     for (i = 0; i < numRptVarNames; ++i)
       {
       varObjNames[i].object_tag = DOM_SPEC;
       varObjNames[i].domain_id = "pbroDev";
       varObjNames[i].obj_name.item_id = rptVarNames[i];
       }

     nvl = mvlu_rpt_nvl_add (&nvlObjName, 5, varObjNames);
```

See page 258 for more information on this function.

3. **Create the MVLU Report Control element**. To do this, you will need to know the variable name for the BASRCB, have created the dataset NVL, and located the Variable Association for the brick that contains the BASRCB variable. In addition, you will need to assign a unique report control ID (typically an integer).

The following function is used to create the MVLU report control:

```
MVLU_RPT_CTRL *mvlu_create_rpt_ctrl (ST_CHAR *basrcbName,
                                     MVL_NVLIST_CTRL *dsNvl,
                                     MVL_VAR_ASSOC *base_va,
                                     ST_INT rcb_type,
                                     ST_INT buftim_action,
                                     ST_INT brcb_bufsize,
                                     ST_UINT32 ConfRev);
```

Note that **`mvlu_create_rpt_ctrl`** does limited initialization of the common BASRCB data (**`DatSetNa`** is intialized, **`OptFlds`** bitstring length is set to 5, **`TrgOps`** is set to `MVLU_TRGOPS_DATA`). All other values are set to 0; any other desired initialization must be done in the application, or by the report client. See page 259 for more information on this function.

An example of the use of this function is as follows:

```
      MVLU_RPT_CTRL *globeStRptCtrl;
      OBJECT_NAME baseVarName;
      MVL_VAR_ASSOC *baseVa;

            baseVarName.object_tag = DOM_SPEC;
            baseVarName.domain_id = "pbroDev";
            baseVarName.obj_name.item_id = "GLOBE";
            baseVa = mvl_find_va (&baseVarName);
      globeStRptCtrl = mvlu_create_rpt_ctrl ("GLOBE$RP$brcbST",nvl,
                                            baseVa,
                                            RCB_TYPE_UCA,
                                            MVLU_RPT_BUFTIM_SEND_NOW,
                                            100000, /*brch_bufsize */
                                            GLOBE_RP_BRCBST_RPT_ID,
                                            0);
```

4. **Attach the MVLU Report Control system to the BASRCB variable primitive elements.** MVLU has read and write handlers for all elements of the BASRCB, and the Foundry template file must be used to select those functions for use with the primitive elements.

   The following is an example of how this is done (in the "TFN" file):

```
/* We will use a set of standard report handling functions      */
#define u_globe_rp_brcbst_rptid_rd_ind_fun      mvlu_rptid_rd_ind_fun
#define u_globe_rp_brcbst_rptena_rd_ind_fun     mvlu_rptena_rd_ind_fun
#define u_globe_rp_brcbst_datsetna_rd_ind_fun   mvlu_datsetna_rd_ind_fun
#define u_globe_rp_brcbst_optflds_rd_ind_fun    mvlu_optflds_rd_ind_fun
#define u_globe_rp_brcbst_buftim_rd_ind_fun     mvlu_buftim_rd_ind_fun
#define u_globe_rp_brcbst_trgs_rd_ind_fun       mvlu_trgs_rd_ind_fun
#define u_globe_rp_brcbst_seqnum_rd_ind_fun     mvlu_seqnum_rd_ind_fun
#define u_globe_rp_brcbst_trgops_rd_ind_fun     mvlu_trgops_rd_ind_fun
#define u_globe_rp_brcbst_rbepd_rd_ind_fun      mvlu_rbepd_rd_ind_fun
#define u_globe_rp_brcbst_intgpd_rd_ind_fun     mvlu_intgpd_rd_ind_fun

#define u_globe_rp_brcbst_rptid_wr_ind_fun      mvlu_rptid_wr_ind_fun
#define u_globe_rp_brcbst_rptena_wr_ind_fun     mvlu_rptena_wr_ind_fun
#define u_globe_rp_brcbst_datsetna_wr_ind_fun   u_no_write_allowed
#define u_globe_rp_brcbst_optflds_wr_ind_fun    mvlu_optflds_wr_ind_fun
#define u_globe_rp_brcbst_buftim_wr_ind_fun     mvlu_buftim_wr_ind_fun
#define u_globe_rp_brcbst_trgs_wr_ind_fun       mvlu_trgs_wr_ind_fun
#define u_globe_rp_brcbst_seqnum_wr_ind_fun     mvlu_seqnum_wr_ind_fun
#define u_globe_rp_brcbst_trgops_wr_ind_fun     mvlu_trgops_wr_ind_fun
#define u_globe_rp_brcbst_rbepd_wr_ind_fun      mvlu_rbepd_wr_ind_fun
#define u_globe_rp_brcbst_intgpd_wr_ind_fun     mvlu_intgpd_wr_ind_fun
```

5. **Create Report Scan Control Elements.** An application can choose to have MVLU scan report data, using the standard MVLU read/write indication functions. To do this, the application creates one or more **MVLU_RPT_SCAN_CTRL** elements using the function **mvlu_rpt_create_scan_ctrl**. Note that the MVLU report scan control elements are independent of the report control elements themselves, and consist primarily of the list of variables to be scanned and associated scan control information. That is, a variable may be used in one or more reports but need be present only once in a scan control element.

   The following is an example of how this is done, where a single scan control element is setup to scan all variables associated with a single UCA report dataset:

```
MVL_NVLIST_CTRL *dsNvl;
MVLU_RPT_SCAN_CTRL *scanCtrl;
ST_INT i;

scanCtrl = mvlu_rpt_create_scan_ctrl (dsNvl->num_of_entries);
for (i = 0; i < dsNvl->num_of_entries; ++i)
  scanCtrl->scan_va[i]= dsNvl->entries[i];
memcpy (scanCtrl->scan_va_scope, nvl->va_scope,
        nvl->num_of_entries * sizeof (MVL_SCOPE));
scanCtrl->enable = SD_TRUE;
```

   After the scan control is created, its control parameters (scan period, enable, etc.) may be modified as desired by the application.

6. **Report Service.** The MVLU function **mvlu_rpt_service** is used to provide MVLU with processing time. This report processing consists of servicing all MVLU Report Scan Control elements, servicing all MVLU Report Control elements (that is, for active clients), and sending UCA reports as appropriate.

The frequency at which **`mvlu_rpt_service`** must be called depends on the accuracy and timeliness required of the system, as well as the data change detection mechanisms selected (see below). For instance, if the application does not make use of the MVLU Report Scan mechanisms and reports changes directly, **`mvlu_rpt_service`** need only be called when data has changed, and at a period suitable for integrity/periodic reports. On the other hand, if the application relies on the scan mechanisms to detect changes, **`mvlu_rpt_service`** should be called frequently enough to not miss data changes that are to be reported.

7. **Data Change Detection.** The application may choose to make use of the MVLU Report Scan Control mechanism to detect data changes, or may report data changes asynchronously as they take place. To do this, the function **`mvlu_rpt_va_change`** is called, with the variable association, new data, and reason for change. MVLU will then buffer the data and track what data has changed.

## *Theory of Operation*

### BASRCB Handling

Each MVLU Report Control element has data storage for BASRCB data. This data is referred to as the "common" data. When a client establishes a connection it can read the BASRCB data, and the source of the data is the "common" BASRCB data. In this state the client is referred to as a "browsing" client; it does not participate in actual report activities.

When the client writes any element of the BASRCB, the client is given it's own BASRCB data storage. From that point on, the client is referred to as an "active" client, and it can control the UCA report mechanisms independent from other clients. An active client can set all BASRCB parameters as desired.

## *Report Dataset Named Variable List Handling*

These functions are used to manage the Named Variable List control elements, which are used in setting up and managing the report control elements. Note that these functions are not necessarily UCA specific, but can also be used to create Named Variable Lists dynamically.

## *Named Variable List Functions*

### mvlu_derive_rpt_ds

**Usage:** This function is used to create the dataset NVL.

**Function Prototype:**
```
MVL_NVLIST_CTRL *mvlu_derive_rpt_ds (ST_CHAR *domName,
                                     ST_CHAR *nvlName,
                                     ST_INT numNodes,
                                     ST_CHAR **nodeNames);
```

**Parameters:**

`domName` This is the name of the domain where the node variables are found and also where the resulting NVL is to be located.

`nvlName` The name of the NVL to be created.

`numNodes` The number of structures from which to derive variable names.

`nodeNames` A table of [**`numNodes`**] pointers to the names of variable nodes from which report variable names will be derived. A typical name will be of the form **“`PBRO$ST`”**, which will cause all members of the structure variable **`PBRO$ST`** to be added as elements of the dataset.

**Return Value:** A pointer to the new Named Variable List object. **`NULL`** if the operation failed. The structure **`MVL_JOURNAL_CTRL`** is defined in **mvl_defs.h**.

## mvlu_rpt_nvl_add

**Usage:** This function to create the report dataset NVL.

**Function Prototype:**

```
MVL_NVLIST_CTRL *mvlu_rpt_nvl_add (OBJECT_NAME *obj,
                                   ST_INT num_var,
                                   OBJECT_NAME *var_obj);
```

**Parameters:**

| | |
|---|---|
| `obj` | This is the MMS Named Variable List name to be used for the report NVL (dataset). |
| `num_var` | The number of variables to be included in the NVL (dataset). |
| `var_obj` | This is an array of MMS Named Variable names to be included in the NVL (dataset). |

**Return Value:** A fully resolved and ready to use MVL Named Variable List control element. NULL will be returned if an error occurs.

## mvlu_rpt_nvl_destroy

**Usage:** This function is used to free a NVL created by **`mvlu_derive_rpt_ds`** or **`mvlu_rpt_nvl_add`**.

**Function Prototype:**

```
ST_VOID mvlu_rpt_nvl_destroy (MVL_NVLIST_CTRL *nvl);
```

**Parameters:**

| | |
|---|---|
| `nvl` | This is pointer to the **`MVL_NVLIST_CTRL`** structure to be freed. |

**Return Value:** A fully resolved and ready to use MVL Named Variable List control element. NULL will be returned if an error occurs.

## *MVLU Report Control Creation Functions*

These functions are used to create and free MVL report control elements. These report control elements are used in the sending of UCA reports and optionally for updating the data to be sent.

### mvlu_create_rpt_ctrl

**Usage:** This function is used to create a MVL Report Control data structure. The input parameter **`prepIndCtrl`** should be set **`SD_TRUE`** if the application will be making use of the MVLU read indication functions to supply data. All report data elements are contained in the DataSet NVL elements and the **`MVLU_RPT_CTRL`** data structure. Note that **`MVLU_RPT_CTRL`** contains a connection oriented set of data structures to provide client applications with their own view of some report control data.

**Function Prototype:**

```
MVLU_RPT_CTRL *mvlu_create_rpt_ctrl (ST_CHAR *basrcbName,
                                     MVL_NVLIST_CTRL *dsNvl,
                                     MVL_VAR_ASSOC *base_va,
                                     ST_INT rcb_type,
                                     ST_INT buftim_action,
                                     ST_INT brcb_bufsize,
                                     ST_UINT32 ConfRev);
```

**Parameters:**

`BasrcbName` This is the variable name for the BASRCB used to control the report. For example, in the GLOBE brick we have **`GLOBE$RP$brcbMX`** and **`GLOBE$RP$brcbST`**.

`dsNvl` This is the dataSet associated with the report control element. This must be a completely resolved Named Variable List. That is, all variable associations must be complete and valid. This will be the case when the dataSet NVL is created using **`mvlu_derive_rpt_ds`** or **`mvlu_rpt_nvl_add`**.

The data pointer input parameters are used to link the MVLU Report Control block with the associated MMS RCB. That is, these references are used as the data source to correspond with the UCA report elements and will typically map to the MMS visible RCB for the report.

`base_va` This is the MVL Variable Association for the brick to which the BASRCB belongs. This can be obtained as shown below:

```
OBJECT_NAME baseVarName;
MVL_VAR_ASSOC *baseVa;

   baseVarName.object_tag = DOM_SPEC;
   baseVarName.domain_id = "pbroDev";
   baseVarName.obj_name.item_id = "GLOBE";
   baseVa = mvl_find_va (&baseVarName);
```

## mvlu_create_rpt_ctrl (con't)

**Parameters (Con't):**

`rcb_type` The Report Control Block type (one of the following):

```
RCB_TYPE_UCA
RCB_TYPE_IEC_BRCB
RBC_TYPE_IEC_URCB
```

`buftim_action` This is the action to be taken of a variable changes twice before the **`BufTim`** timer expires (one of the following):

```
MVLU_RPT_BUFTIM_REPLACE
MVLU_RPT_BUFTIM_SEND_NOW
```

`brbc_bufsize` This is the maximum amount of memory (in bytes) to allow for storing IEC-61850 Buffered Reports. It is used only if **`rcb type = RCB_TYPE_IEC_BRCB`**.

`ConfRev` This is the value to send to clients in a read response when the **`ConfRev`** parameter is read from the RCB.

**Return Value:** This function returns the MVL report control element, **`MVLU_RPT_CTRL`**. NULL will be returned if an error occurs.

## mvlu_free_rpt_ctrl

**Usage:** This function is used to free a MVLU Report Control element created via **`mvlu_create_rpt_ctrl`**.

**Function Prototype:** `ST_VOID mvlu_free_rpt_ctrl (MVLU_RPT_CTRL *rptCtrl);`

**Parameters:**

`rptCtrl` A pointer to a **`MVLU_RPT_CTRL`** structure to be freed.

**Return Value:** `ST_VOID`

## Report Variable Scanning Functions

### mvlu_rpt_create_scan_ctrl2

**Usage:** This function is used to create a **`MVLU_RPT_SCAN_CTRL`** element. Note that the MVLU report scan control elements are independent of the report control elements themselves and consist primarily of the list of variables to be scanned and associated scan control information. That is, a variable may be used in one or more reports but need be present only once in a scan control element.

**Function Prototype:**

```
MVLU_RPT_SCAN_CTRL *mvlu_rpt_create_scan_ctrl2 (MVL_NVLIST_CTRL *nvl,
                                              ST_RET (*scan_done_fun)
                                              (struct mvl_ind_pend *ind_pend),
                                              ST_UINT report_scan_rate);
```

**Parameters:**

| | |
|---|---|
| `nvl` | Report DataSet (Named Variable List) to scan. |
| `scan_done_fun` | Pointer to optional user function to be called when each scan completes. NULL if user function not needed. |
| `report_scan_rate` | Report scan rate (in milliseconds). |

| | | |
|---|---|---|
| **Return Value:** | `!= NULL` | Returns a pointer to a **`MVLU_RPT_SCAN_CTRL`** structure. |
| | `= NULL` | An error has occurred. |

### mvlu_rpt_create_scan_ctrl

**Usage:** This function is used to create a **`MVLU_RPT_SCAN_CTRL`** element. Note that the MVLU report scan control elements are independent of the report control elements themselves and consist primarily of the list of variables to be scanned and associated scan control information. That is, a variable may be used in one or more reports but need be present only once in a scan control element.

***NOTE:*** **`mvlu_rpt_create_scan_ctrl2`** performs more initialization of the **`MVLU_RPT_SCAN_CTRL`** structure, so it should be easier to use in most cases.

**Function Prototype:**

```
MVLU_RPT_SCAN_CTRL *mvlu_rpt_create_scan_ctrl (ST_INT numScanVa);
```

**Parameters:**

| | |
|---|---|
| `numScanVa` | The number of variables to scan. |

| | | |
|---|---|---|
| **Return Value:** | `!= NULL` | Returns a pointer to a **`MVLU_RPT_SCAN_CTRL`** structure. |
| | `= NULL` | An error has occurred. |

## *Report Service Functions*

### mvlu_rpt_service

**Usage:** This function is used to provide MVLU with processing time. This report processing consists of servicing all MVLU Report Scan Control elements, servicing all MVLU Report Control elements (that is, for active clients), and sending UCA reports as appropriate.

**Function Prototype:**

```
ST_VOID mvlu_rpt_service (ST_VOID);
```

**Parameters:** NONE

**Return Value:** ST_VOID

## *Asynchronous Change Reporting Functions*

### mvlu_rpt_va_change

**Usage:** This function is used to make use of the MVLU Report Scan Control mechanism to detect data changes, or may report data changes asynchronously as they take place. To do this, it is called with the variable association, new data, and reason for change. MVLU will then buffer the data and track what data has changed.

**Function Prototype:**

```
ST_VOID mvlu_rpt_va_change (MVL_VAR_ASSOC *va,
                            ST_UCHAR reason,
                            ST_VOID *new_data);
```

**Parameters:**

| | |
|---|---|
| va | This is a pointer to a **MVL_VAR_ASSOC** structure containing the variable association information. |
| reason | This contains the reason for the change. |
| new_data | This contains the newly changed data. |

**Return Value:** ST_VOID

### *Lower Level Functions*

These functions may be used to take direct control of the scanning process. For usage information refer to the MVL source code.

## mvlu_rpt_va_scan

**Usage:** This function is called by **`mvl_rpt_service`** to scan all UCA Report variables.

**Function Prototype:** `ST_VOID mvlu_rpt_va_scan (ST_VOID);`

**Parameters:** NONE

**Return Value:** `ST_VOID`

## mvlu_rpt_scan_read

**Usage:** This function is called by **`mvlu_rpt_va_scan`** to scan an individual group of variables.

**Function Prototype:**

```
ST_VOID mvlu_rpt_scan_read (MVLU_RPT_SCAN_CTRL *scanCtrl);
```

**Parameters:**

`scanCtrl` Pointer to the scanning control structure that was created by **`mvlu_rpt_create_scan_ctrl`**.

**Return Value:** `ST_VOID`

# MVL UCA SBO Handling

MVLU contains a set of functions and data structures that are useful in handling the UCA Select Before Operate (SBO) features. This support comes in the form of common UCA Read/Write Indication functions that are attached to the SBO element and the protected object, and a SBO control data structure. Please see the sample server source, **uca_srvr.c**, for an example of the use of these functions.

```
typedef struct
  {
  ST_BOOLEAN in_use;                  /* control element management */
  ST_CHAR sbo_var[MAX_SBO_NAME_SIZE+1];
  MVL_NET_INFO *net_info;             /* Connection ID              */
  time_t expire_time;                 /* SELECT expiration time     */
  } MVL_SBO_CTRL;
```

The following user defined function is called to operate the protected element.

```
ST_VOID u_mvl_sbo_operate (MVL_SBO_CTRL *sboSelect,
                           MVLU_WR_VA_CTRL *mvluWrVaCtrl);
```

The following function is used to terminate any pending SBO operations on the selected connection and is typically called when a connection is terminated.

```
ST_VOID mvlu_clr_pend_sbo (MVL_NET_INFO *net_info);
```

## *UCA SBO Read/Write Indication Handler Functions*

These Read/Write indication handler functions are to be attached to the appropriate SBO objects via the Foundry template input file.

```
      ST_VOID mvlu_sbo_operate_wr_ind (MVLU_WR_VA_CTRL *mvluWrVaCtrl);
      ST_VOID mvlu_sbo_select_rd_ind (MVLU_RD_VA_CTRL *mvluRdVaCtrl);
```

# MVL_UCA Compilation Options

To make use of the MMS-EASE *Lite* UCA extensions, **`MVL_UCA`** must be defined during compilation of the core MMS libraries, MVL libraries, and all user code. This define works to enable the enhanced **`RUNTIME_TYPE`** features required for handling the UCA object models effectively.

Note that by default this is defined automatically when **`MMS_LITE`** is defined, and is fully compatible with non-UCA applications.

To implement the UCA support subsystem, MVLU creates Runtime Types dynamically and the MVL type control table is allocated at initialization time. As a result, MVL must know the maximum number of active “dynamic” types at any given time. This is done via the following define:

```
      #define MVLU_NUM_DYN_TYPES    100
```

This define controls the number of “dynamic” MVL type control elements available for use by MVLU. This should be set to the max number of variables per read * the number of indications pending.

## *SBO Control Defines*

```
#define SBO_SELECT_TIMEOUT      30    /* seconds                    */
#define MAX_NUM_SBO_PEND        10    /* Number of SBOs to be pending */
```

## Chapter 8

# MMS Object Foundry

MMS Object Foundry is a MMS-EASE *Lite* tool for creating MMS server objects, including Types, Variables, Named Variable Lists, and Domains. MMS Object Foundry must be used for all MMS-EASE *Lite* applications that make use of MVL. Its use greatly simplifies the process of creating links between MMS objects such as variables and local program variables or processes. In addition, MMS Object Foundry has UCA device model specific features that make implementing such devices a straightforward task.

MMS Object Foundry is generally run on the command line or in a makefile and is supplied in source form as well as Windows NT/2000/XP executable forms.

## MMS Object Foundry Workflow

As shown below, the primary function of MMS Object Foundry is to take a MVL Object Definition File (text file) as input and create a C source code module and associated header file as output. These output files are then compiled and linked to the MMS-EASE *Lite* application, where they provide code to initialize all of the defined MMS objects automatically. The additional input file **align.cfg** is used to tell MMS Object Foundry the alignment requirements of the target compiler for data structure member alignment.

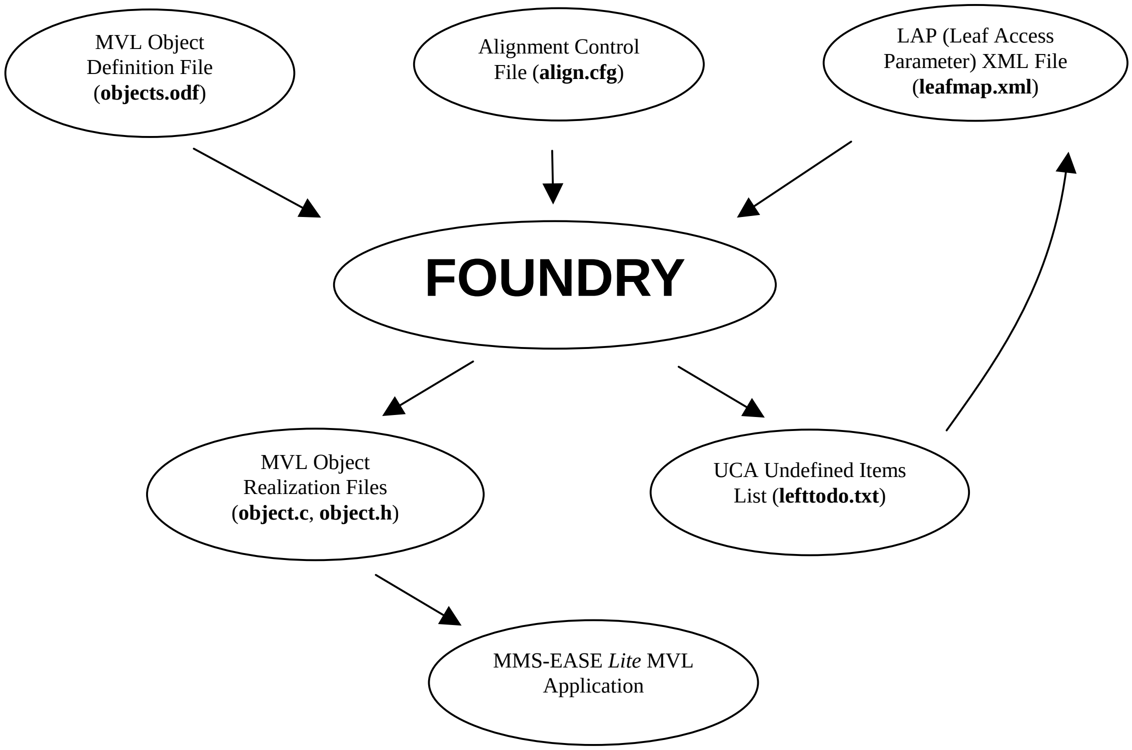

MVL Object Definition File (**objects.odf**)
Alignment Control File (**align.cfg**)
LAP (Leaf Access Parameter) XML File (**leafmap.xml**)
FOUNDRY
MVL Object Realization Files (**object.c**, **object.h**)
UCA Undefined Items List (**lefttodo.txt**)
MMS-EASE *Lite* MVL Application

Figure 12: MMS Object Foundry Workflow Diagram

In the simple case, this is done with the following steps.

1. Review the alignment control file for correctness.
2. Create an Object Definition File
3. Run MMS Object Foundry with the ODF and alignment control file as input.
4. Compile the resulting C file and link to your application.

***Note:*** *No user modification to the output C file is needed or desirable; the only source that is to be edited is the Object Definition File and the Alignment Control File.*

# Command Line Parameters

MMS Object Foundry uses the following command line syntax:

```
foundry [options] [-calignFile] [-tlapXmlFile] objectFile [outputFile]
```

where:

| | |
|---|---|
| `-c{alignFile}` | specifies the structure alignment input file |
| `-t{lapXmlFile}` | specifies the LAP (Leaf Access Parameter) XML input file |
| `objectFile` | specifies the Object Definition input file |
| `outputFile` | specifies the name of the output file |

and optionFlags are one or more of the following:

| | |
|---|---|
| [`-o`] : | Overwrite target |
| [`-n`] : | Extract UCA Variable Names |
| [`-v`] : | Create UCA Variable Names & Associations |
| [`-p`] : | Print line numbers being processed |
| [`-d`] : | Debug mode |

## LAP XML Input File

This file, specified with the "-t"command line option, contains "**Leaf Access Parameter**" information to map primitive data elements to "leaf" functions and "references" for IEC-61850 or UCA variables. The file name must contain the extension "**.xml**". If the file name does NOT contain the extension "**.xml**", it is assumed to be a "Template File" (described later). The "Template File" should **NOT** be used because it is much more complicated. It  remains an option only for backward compatibility.

Below is a very simple example of a “LAP XML input file”:

```
<Leafmap>
 <Leaf Name="DI$Name" RdIndFun="rdString" WrIndFun="noWrite" Ref="42"/>
 <Leaf Name="DI$Class" RdIndFun="rdString" WrIndFun="noWrite" Ref="43"/>
</Leafmap>
```

- The **`Name`** attribute contains the name of the leaf.
- The **`RdIndFun`** attribute contains the name of the “Read leaf function”.
- The **`WrIndFun`** attribute contains the name of the “Write leaf function”.
- The **`Ref`** attribute contains any text that may be used as the “reference” value (the **`ref`** element of the **`RUNTIME_TYPE`** structure) in the output “C” file. The “reference” is the parameter that is passed in the “primRef” element of the **`MVLU_RD_VA_CTRL`** structure (passed to “read” leaf functions) or the **`MVLU_WR_VA_CTRL`** structure (passed to “write” leaf functions). For example, if the XML file contains the following **`Ref`**:

  ```
  Ref=”&user_global_var”
  ```

  the following line is generated in the appropriate **`RUNTIME_TYPE`** structure in the output “C” file:

  ```
  &user_global_var               /*   ref                  */
  ```

**NOTE:** The same XML file may be passed to the function **`mvlu_load_xml_leaf_file`** to load the parameters at runtime.

**IMPORTANT:**

1. Foundry also generates a “**LAP XML output file**” named **lap_out.xml**. It follows exactly the same format as the “LAP XML input file”. However, it contains “dummy” entries for any leafs that are not mapped in the input file. It may easily be edited to replace the “dummy” entries with correct entries, then used to replace the “LAP XML input file”.
2. If the “LAP XML Input File” (not the “Template File”) is used as input to Foudry, the **lefttodo.txt** output file generated by Foundry contains an easy to read summary of missing Leaf Access Parameters (instead of starter functions).

# Output File

Note that the **outputFile** parameter is used (minus extension) to create the filenames to be written as output files, with the **.c** and **.h** extensions. If outputFile is omitted, the output files names will be derived from the objectFile name.

# Alignment Control File

This file is used to help MMS-EASE *Lite* map complex data types onto local C data types. It specifies the data alignment requirements of the various primitive and complex data elements. See *Alignment Control File* on page 275 for more information on creating and maintaining this file.

## The Object Definition File

The primary input to MMS Object Foundry is the Object Definition File. This is a text file with notation for creating MMS objects and controlling their attributes easily and automatically.

Some general features of the Object Definition File are:

- Anything following the # character is treated as a comment and ignored.
- Blank lines are ignored.
- All object information is contained in quotes.
- Object Definition Strings can span multiple lines.
- Object Definition Files can "include" other Object Definition Files.

## Including Object Definition Files

ODF files can "include" other ODF files, which allows reuse of Object Definitions as building blocks. This is done with the following syntax:

```
include xxxxx
```

where `xxxxx` is the file to be included. Note that the keyword "include" is not in quotes and must be the first word on a line, followed by one space and then the file to be included.

# Object Definition Syntax

In general, an object definition string is of the following form:

```
"{Object Type}{Object Flags)","Object Specific String1", "String2", ...
```

Where the Object Type is a single character used to identity the type of object being defined, the Object Flags is one or more characters used to specify some processing attribute for the object, and the Object Specific Strings are used to specify the attributes of the object.

| Object | Object Type Code | Object Option Flags |
|---|---|---|
| MMS Object Foundry Execution Control | C | F, C, U, P |
| User Include File | I | |
| MMS Data Type | T | K, U, T, V, X |
| MMS Domain | D | |
| MMS Variable | V | D, P, U |
| MMS Named Variable List | L | |
| UCA Name Generation | N | |

# MMS Object Foundry Execution Control

These objects do not result in MMS object creation, but rather provide control over the creation of subsequent objects. The form of this object definition is as follows:

```
":C{flags}", "ControlArgument"
```

| Parameter | Description |
|---|---|
| Control Argument | The possible values and effects of this parameter depend on the flags used. See the following chart for valid control arguments. |

| Flag Character | Attribute |
|---|---|
| P | This parameter is used to identify an include path for MMS Object Foundry Object Definition Files. Multiple paths may be specified. MMS Object Foundry will try to open the include file in the working directory and then in each of the specified path directories. In this case, the ControlArgument string is the path to be searched. |
| C | This parameter is used to select the alignment control file to be used. In this case, the ControlArgument string is the filename for the alignment control file. |
| F | This flag is used to override the default attributes for the Named Variable and Data Type object types. The valid ControlArgument string values are:<br>:V{DP}<br>:T{UTBV}<br>The attributes that are present for the object type will then be applied to subsequent objects of that type. Attributes that are not present will revert to default values. See *MMS Data Types* on page 270 and *MMS Variable Objects* on page 271 for attribute value descriptions. |
| U | This parameter is used to control UCA/IEC-61850 specific processing by MMS Object Foundry. The valid values for ControlArgument are:<br>`MVL_UCA` - enables MVL UCA/IEC-61850 processing<br>`MVLU_USE_REF` - enables use of references<br>See *MMS Object Foundry UCA Specific Features* on page 276 for more information on the use of these parameters and other UCA support issues. |

# User Include File

This object type is used to specify files that are to be included in the output C file. This is necessary when creating variables with data or processing initialization strings. The form of this object definition is as follows:

```
":I", "fileName"
```

| Parameter | Description |
|---|---|
| fileName | This string will be placed in an #include statement output C file. |

An example of this is to include the user header file **srvrobj.h** in the object definition.

```
":I", "srvrobj.h"
```

This will result in the following line being placed in the output C file:

```
#include "srvrobj.h"     /* User Specified */
```

# MMS Data Type

This section is used to create MMS-EASE *Lite* data types. The form of this object definition is as follows:

```
":T{flags}", "TypeId", "TDL", "Comment"
```

| Parameter | Description |
|---|---|
| TypeId | This string will be used to create the define to be used to reference the type in the application. |
| TDL | This is the SISCO type definition language string, which defines the type. See Type Description Language on page 371. |
| Comment | This string is a text comment used only in the output C file. |

| Flag Character | Attribute |
|---|---|
| K | Keep this type unconditionally. |
| U | Unique type. |
| T | Transient type; discard after processing. Note that transient types are to be used only in constructing more complex types. |
| V | Discard this type unless referenced by a configured variable. |
| X | UCA/IEC-61850 type. Perform additional initialization for UCA/IEC-61850 (i.e. set leaf functions and references) even if there are no UCA/IEC-61850 variables using this type. **Important if UCA/IEC61850 variables created at runtime**. |

For example, to create a MMS data type for a simple structure like the following:

```
typedef struct struct1
      {
      ST_INT16    s;
      ST_INT32    l;
      } STRUCT1;
```

Use the following configuration element:

```
":T", "STRUCT1_TYPE","{(s)Short,(l)Long}", "Basic simple structure"
```

# MMS Domain

This section is used to create MMS-EASE *Lite* domains. The single parameter is the domain name, which must be a legal MMS domain name. The form of this object definition is as follows:

```
":D", "domName"
```

To create a domain named mvlLiteDomain, use the following command:

```
":D", "mvlLiteDomain"
```

| Parameter | Description |
|---|---|
| DomainName | Name of the MMS domain to create. |

# MMS Named Variables

This section is used to create MMS-EASE *Lite* MVL Variable Associations, which are the MVL control element instantiated MMS Named Variables. The form of this object definition is as follows:

```
":V{flags}", "VarName", "TypeId", {"Data"}, {"ProcFuns"}
```

| Parameter | Description |
|---|---|
| VarName | Name of MMS variable, which must conform to MMS naming conventions: 1-32 characters, valid characters being [a-zA-Z0-9$_], and must not start with a digit. The scope selector may optionally prefix the VarName. For instance, “domName:VarName” will result in a variable named “VarName” belonging to the scope of the domain named “domName”. Application Association scope is selected by using the prefix “AA_SCOPE:”. When no scope selector prefix is used, the scope is VMD. |
| TypeId | Type ID for MMS data type. Must have been previously defined. |
| Data | This optional string is used to initialize the va->data element in a MVL Variable Association. To use this option, the D flag must be present. Note that elements referenced by the Data string must typically be resolved by using an Include directive. |
| ProcFuns | This optional string is used to initialize the va->proc element in a MVL Variable Association. To use this option, the P flag must be present. Note that elements referenced by the ProcFuns string must typically be resolved by using an Include directive. |

| Flag Character | Attribute |
|---|---|
| D | When present, the 3rd string is used to initialize the va->data element. |
| P | When present, the 4th string (or 3rd if no D flag) is used to initialize the va->procFuns element. |
| U | UCA variable. When present, the 3rd string is used to initialize the va->user_info element. |

# MMS Named Variables Examples

## *Example 1*

Use the following command to create a VMD scope variable named **`myStructVar`** of type `STRUCT1` that maps to a local variable **`STRUCT1 myStructVar`**. In addition, use pre/post processing selected by **`MVL_VAR_PROC varProcFuns`**.

```
":VDP", "myStructVar", "STRUCT1_TYPE", "&myStructVar", "&varProcFuns"
```

## *Example 2*

Use the following command to create a VMD scope variable named **`myStructVar`** of type `STRUCT1` that maps to a local variable **`STRUCT1 myStructVar`**.

```
":VD", "myStructVar", "STRUCT1_TYPE", "&myStructVar"
```

## *Example 3*

Use example 2 but create a domain scope belonging to the domain **`mvlLiteDomain`**.

```
":VD", "mvlLiteDomain:myStructVar", "STRUCT1_TYPE", "&myStructVar"
```

## *Example 4*

Define an Application Association scope variable named **`reportControl`** that maps onto an element of a local array of 16 bit integers. Note the use of the `[i]` array index in the data string. This works to associate the elements of the variable array with the MVL called connection control array **`mvl_called_conn_ctrl`**.

```
":VD", "AA_SCOPE:reportControl", "INTEGER16_TYPE", "&reportControl[i]"
```

## *Example 5*

Create a domain specific UCA variable called “MU” belonging to the domain **`ln0`**.

```
":VU", "ln0:_UCA_MU", "MU", "1"
```

# MMS Named Variable List

This section is used to create MMS-EASE *Lite* Named Variable Lists. A Named Variable List is essentially a list of previously defined Named Variables. The form of this object definition is as follows:

```
":L", "VarListName", "VarName", {"VarName"} ... , ":S"
```

| Parameter | Description |
|---|---|
| VarListName | Name of MMS Named Variable List, must conform to MMS naming conventions: 1-32 characters, valid characters being [a-zA-Z0-9$_], and must not start with a digit. The scope selector may optionally prefix the VarListName. For instance, `domName:VarListName` will result in a Named Variable List named “VarListName” belonging to the scope of the domain named `domName`. Application Association scope is selected by using the prefix `AA_SCOPE`. When no scope selector prefix is used, the scope is VMD. |
| VarName | This is a sequence of strings selecting Named Variables to be included in the NamedVariableList. These VarNames must have all scope information included and must be defined previously. Note that the scopes of all items are independent. For instance, a VMD scope Named Variable List can reference variables from VMD, domain, or AA scopes.<br>**UCA Note:** When the `MVL_UCA` mode is enabled, these variables do not need to be defined as they may be derived. |
| :S | This string functions as an “end of variables” marker. |

## MMS Named Variable List Examples

### Example 1

Use the following command to create a VMD scope Named Variable List named **nvl1** that contains the Named Variables **arr1** and **Temperature**.

```
":L", "nvl1", "arr1", "Temperature", ":S"
```

### Example 2

Use the following command to create a Domain scope NamedVariableList named **nvl1** that contains the Named Variables **domArr1** and **domTemperature**. All elements belong to the domain **mvlLiteDom**.

```
":L", "mvlLiteDom:nvl1",  "domArr1",  "mvlLiteDom:domTemperature",  ":S"
```

# UCA Model Name Generation

MMS Object Foundry can generate UCA model form variable names from a selected structure type. This naming convention uses the $ symbol as a structure nesting delimiter and provides alternate views of the structure at all levels. Note that MMS Object Foundry will only do name generation separately from its normal mode of operation; it will not generate standard object realization code at the same time. A command line switch is used to toggle MMS Object Foundry modes. The form of this object definition is as follows:

```
":N", "BaseName", "TypeId"
```

The first parameter is the name base to be used. The second parameter is the type to use in extracting the names and will generally be a high level UCA object type.

***Note:*** *These objects are used only when the* **`–v`** *or* **`-n`** *command line parameter is used.*

| Parameter | Description |
|---|---|
| BaseName | This is the base name to be used in generating the variable names. |
| TypeId | Type ID for MMS data type. Must have been previously defined. |

As an example, derive UCA device model names from the data structure type **`STRUCT1_TYPE`** using **`struct1`** as the base variable name. **`STRUCT1_TYPE`** has been defined as:

```
":T", "STRUCT1_TYPE","{(s)Short,(l)Long}", "Basic simple structure"
":N", "struct1", "STRUCT1_TYPE"
```

The output would appear as follows:

```
struct1
struct1$s
struct1$l
```

# Alignment Control File

The contents of the alignment control file are used to tell MMS Object Foundry how data is stored in memory by the C compiler. The idea is that addresses of the data types described in the table cannot have bits set that are set in the table values.

For instance, if the value is 0x0000, the corresponding data type can be on any memory boundary. If it is 0x0001, it must be on even work boundary.

SISCO supplies standard data alignment files for DOS, WIN32, and QNX environments. Others may be created by reading the compiler alignment rules, or by examining the source file **mms_tdef.c** for the desired environment's table. Alternatively, the SISCO utility program **findalgn** may be compiled and executed in the target environment and will output an appropriate alignment control file.

The contents of the alignment control file are as follows:

```
ST_INT m_def_data_algn_tbl[NUM_ALGN_TYPES] =
  {
  0x0000,   /* ARRSTRT_ALGN   00  */
  0x0000,   /* ARREND_ALGN    01  */
  0x0003,   /* STRSTRT_ALGN   02  */
  0x0000,   /* STREND_ALGN    03  */
  0x0000,   /* INT8_ALGN      04  */
  0x0001,   /* INT16_ALGN     05  */
  0x0003,   /* INT32_ALGN     06  */
  0x0007,   /* INT64_ALGN     07  */
  0x0003,   /* FLOAT_ALGN     08  */
  0x0007,   /* DOUBLE_ALGN    09  */
  0x0000,   /* OCT_ALGN       10  */
  0x0000,   /* BOOL_ALGN      11  */
  0x0000,   /* BCD1_ALGN      12  */
  0x0001,   /* BCD2_ALGN      13  */
  0x0003,   /* BCD4_ALGN      14  */
  0x0000,   /* BIT_ALGN       15  */
  0x0000    /* VIS_ALGN       16  */
  };
#define M_STRSTART_MODE          M_STRSTART_MODE_LARGEST
#define M_STREND_MODE            M_STREND_MODE_LARGEST
```

# MMS Object Foundry UCA Specific Features

## MVL_UCA Overview

The MMS-EASE *Lite* “UCA Extensions” (MVLU) is a run time installable subsystem for MVL that makes handling the complex UCA device models significantly easier and more efficient. By making use of this package, the developer can rely on MVL to generate and support all variable names and permutations, handle MMS Alternate Access transparently, and provide an easy to use mechanism to associate the UCA MMS variables to the real application data.

The general MVLU processing model is that there is one or more high level “base types” present in the device model. A base type is the highest level object accessible and is made up of a set of “sub-types”, which present subset views of the base type; this is similar in nature to the MMS alternate access mechanisms. The base MMS UCA variables are defined using the base types. MVLU then derives the sub-variables from the base type and allows the user application to deal only with the primitive data elements.

When UCA processing is enabled, the MMS-EASE *Lite* Runtime Type is enhanced to support specialized processing and MMS Object Foundry generates code to initialize these new elements. MMS Object Foundry identifies UCA Variables and NamedVariableLists via a naming convention. For each UCA data type, MMS Object Foundry provides Read and Write Indication functions for each primitive elements of the type, as well as a user controlled Reference handle for the primitive element.

Note that one-to-one local application variables for the UCA variables need not exist; `MVL_UCA` provides all required mapping and buffer management to correctly support the UCA model for the device.

In order to use MVLU to provide UCA object support, the user application must provide functions to provide access to the primitive data that collectively makes up the device object. MVLU generates starter code for all required user functions. The two types of functions used are:

Read Indication: Used to handle MMS Read indications

Write Indication: Used to handle MMS Write indications

The starter code for these functions is written to **lefttodo.txt** and is to be edited to become an input Template File.

MMS Object Foundry can also generate reference elements to be associated with the primitive elements of a type. This reference allows the application to use an indication function to service multiple elements of the type, thereby reducing coding effort and code size. The reference is of type `ST_RTREF`, which by default is defined as `ST_VOID *`. This can be changed as necessary for the application indication functions. This typedef is in the MMS-EASE *Lite* header file **mms_vvar.h**.

***WARNING:*** *If the typedef is changed, all the MMS-EASE Lite source modules must be recompiled so the new definition is used consistently.*

# MMS Object Foundry Workflow for UCA Devices

For UCA devices, in order to realize the application's MMS objects, the main objective of MMS Object Foundry is to take your Object Definition File (a text file) and a UCA Function/Define Template File, and produce a C output file that is linked to your application.

To accomplish this, the following process is used:

1. Review the Alignment Control File (ACF) for correctness.
2. Create an Object Definition File (ODF), which will reference the UCA model definitions.
3. Run MMS Object Foundry with the ODF and ACF as input.
4. Take the resulting **lefttodo.txt** file and use as the start of the Template Input file.
5. Implement the Read/Write indication functions found in the Template Input File.
6. Run MMS Object Foundry with the ODF, Template, and alignment control file as input.
7. Compile the resulting C file and link to your application.

Note that no user modification to the output C file is needed or desirable. The only source that is to be edited is the Object Definition File, the UCA Function Template file, and the Alignment Control File.

# UCA Model Object Definition Files

SISCO provides a set of Object Definition Files for UCA objects with MMS-EASE *Lite*. These ODFs contain the fully expanded UCA data type definitions for the following UCA models:

| |
|---|
| Switch (Sw) |
| Switch Controller (SwC), |
| Automatic Switch Controller (ASwC), |
| Breaker (Bkr) |
| Breaker Controller (BkrC) |
| Time Delay Starting or Closing Relay |
| Checking of Interlocking Relay |
| Voltage per Hertz Relay |
| Directional Power Relay |
| Under Current or Under Power Relay |
| Reverse Phase or Phase Balance Current Relay |
| Incomplete Sequence Relay |
| Machine, Transformer Thermal Relay |
| Instantaneous Overcurrent Relay |
| Voltage or Current Balance Relay |

| |
|---|
| Time Delay Starting or Stopping Relay |
| Alarm Relay |
| Phase Angle Measuring Relay |
| Frequency Relay |
| Carrier or Pilot-wire Relay |
| Lockout Relay |
| Tripping or Trip Free Relay |
| Closing Relay/Contactor |
| XYZ Auxiliary Relays |
| Under Voltage Relay |
| Over Voltage Relay |
| Time Overcurrent Relay |
| Distance Relay |
| Sync Relay |
| High Impedance Ground Detector Relay |
| Directional Overcurrent Relay |
| Reclosing Relay |
| Differential Relay |
| Generic Object Oriented Substation Event |
| Capcitor Bank |
| Measuring Unit |

# Enabling MMS Object Foundry UCA Processing

To allow MMS Object Foundry to generate UCA specific code, the following control option must be present in the Object Definition File.

```
":CU", "MVL_UCA"
```

Most UCA applications will want to use reference handles for the primitive data elements and should also include the following control option:

```
":CU", "MVLU_USE_REF"
```

In addition, the following control object should normally be included, allowing MMS Object Foundry to discard all intermediate types used in creating the UCA data types.

```
":CF", ":TV"            # Delete types not used by variables
```

These settings will result in the most effective implementation and will allow MMS Object Foundry to discard all types that are not used directly by configured variables.

To suppress generation of placeholder functions and references, include the line:

```
   ":CU", "MVLU_NO_DUMMY"
```

in the Object Definition (ODF) file. With this parameter, Foundry will no longer generate separate placeholder read/write indication functions and references for each leaf not covered in the "Template" (TFN) input file. Instead, a default 'null' function will be used, either **`mvlu_null_rd_ind`** or **`mvlu_null_wr_ind`**, which will allow linking and execution.

**NOTE:** This flag is not necessary when using the "LAP XML Input File", as Foundry will not generate placeholder functions and references in this case. It is only used when the "Template File" (TFN) is used as input.

# UCA Named Variable Handling

UCA Variables are configured with the prefix **`_UCA_`**. The variable name (minus prefix) is used as the base name for all derived UCA variables.

The following are examples of how to create a UCA Instantaneous Overcurrent Relay variable and all its derived variables. They show one in the VMD scope and one in domain scope.

```
":V","_UCA_IOC","BRO"
":V","ucaDomain:_UCA_IOC","BRO"
```

# NamedVariableList Handling

UCA Named Variable Lists are also configured with the prefix `"_UCA_"`. The name (minus prefix) is the exposed MMS name. For UCA NVLs, the variables in the list need not be defined as configured variables as they are assumed to contain manufactured variables such as the derived UCA variables.

# Miscellaneous Foundry Features

- **Generates Leaf Function-Name to Function-Address Lookup Tables**

If **`MVLU_LEAF_FUN_LOOKUP_ENABLE`** is defined (preferably in **glbopt.h**), Foundry generates "2-dimensional" leaf function tables for the read and write indication "leaf" functions. These tables contain function-name to function-address lookup information, which is required by the dynamic type creation functions **`mvlu_set_leaf_param_name`** and **`mvlu_load_xml_leaf_file`**, because they must search for the leaf functions by name.

Chapter 9

# IEC GOOSE and GSSE Support

## General GOOSE Information

The Generic Object Oriented Substation Event (GOOSE), as defined by the Generic Object Models for Substation & Feeder Equipment (GOMSFE) and IEC 61850, is implemented as a MMS InformationReport sent to multiple destinations using the multicast capability of the Ethernet MAC layer. The data to be sent in the InformationReport, as well as the timing is defined in GOMSFE or IEC 61850.

MMS-EASE Lite supports high performance GOOSE message handling. This support consists of a high performance threadsafe GOOSE message encode/decode system and a low-level function call API for sending and receiving GOOSE messages. All GOOSE message timing and content issues are to be handled by the application programmer. This feature uses the SISCO GLBSEM subsystem for event handling, mutex semaphore support, and thread starting. The GLBSEM subsystem may need to be ported to the target operating system.

***Note:*** *GOOSE processing may occur in one or more separate threads independent of the main MMS communications service thread, which allows the use of high priority threads for GOOSE handling.*

## Subnetwork functions used for GOOSE Support

The following Subnetwork functions are used for supporting IEC GOOSE and UCA GOOSE. Please refer to *Appendix G* on page 363 for detailed information on these functions.

**`clnp_snet_read`** - this function reads a PDU from the subnetwork.

**`clnp_snet_free`** - this function frees up subnetwork resources associated with a received PDU.

**`clnp_snet_set_multicast_filter`** (or **`gse_set_multicast_filter`**) - This function enables the reception of multicast packets (including GOOSE messages) by the Ethernet driver.

Additional functions are provided to assist in finding all available sources of GOOSE messages on a sub-network (i.e. "GOOSE Discovery Mode"). These functions may be very useful in applications where the user dynamically chooses the GOOSE messages to accept.

**clnp_snet_rx_all_multicast_start** (or **gse_discovery_start**) - This function enables the reception of "ALL multicast" packets by the Ethernet driver so that ALL incoming multicast packets (including GOOSE messages) are accepted. The driver remains in this mode until **clnp_snet_rx_all_multicast_stop** is called. When a GOOSE message is received, the destination MAC address may be compared to the "subscribed" addresses (i.e., the addresses passed to **clnp_snet_set_multicast_filter**), to determine if it is a "subscribed" GOOSE or a "discovery" GOOSE.

**clnp_snet_rx_all_multicast_stop** (or **gse_discovery_stop**) - This function disables the reception of "ALL multicast" packets by the Ethernet driver. It will continue accepting multicast packets (including GOOSE messages) that were "subscribed" for using **clnp_snet_set_multicast_filter**.

The following macros are provided for more consistent naming:

```
#define gse_set_multicast_filter    clnp_snet_set_multicast_filter
#define gse_discovery_start         clnp_snet_rx_all_multicast_start
#define gse_discovery_stop          clnp_snet_rx_all_multicast_stop
```

# IEC GOOSE

## IEC GOOSE Decode Data Structures

```
typedef struct
  {
  ST_CHAR *dataRef;
  ST_INT32 elementId;
  ST_UINT8 *asn1Ptr;
  ST_INT asn1Len;
  } GSE_IEC_DATA_ENTRY_RX;

typedef struct
  {
  ST_CHAR *gcRef;
  ST_UINT32 timeToLive;
  ST_CHAR *dataSetRef;
  ST_CHAR *appID;
  MMS_UTC_TIME utcTime;
  ST_UINT32 stNum;
  ST_UINT32 sqNum;
  ST_BOOLEAN test;
  ST_INT32 confRev;
  ST_BOOLEAN needsCommissioning;
  ST_INT8 sendMode;
  ST_INT32 numDataEntries;
  ST_INT tmpIndex; /* index to current entry in "dataEntries" array.  */
                   /* Used during decode when filling in"dataEntries".*/
  GSE_IEC_DATA_ENTRY_RX *dataEntries; /* ptr array data entry structs */
  } GSE_IEC_HDR;
```

# IEC GOOSE Decode Functions

The IEC GOOSE decode is invoked by the user application. The complete decode requires calling at least three functions, **`gse_iec_hdr_decode`**, **`ms_asn1_to_local`**, and **`gse_iec_decode_done`**.

## *Header Decode Function*

### gse_iec_hdr_decode

**Usage:** This function decodes the header of a IEC GOOSE message, but not the data.

**Function Prototype:**

```
GSE_IEC_HDR *gse_iec_hdr_decode (SN_UNITDATA *sn_udt);
```

**Parameters:**

| | |
|---|---|
| `sn_udt` | A pointer to a **`SN_UNITDATA`** structure containing the input IEC GOOSE message. |

**Return Value:**

| | |
|---|---|
| `(GSE_IEC_HDR *)` | A pointer to a structure allocated by the function, containing the output (i.e., decoded) IEC GOOSE header data. This pointer must be freed when it is no longer in use, by calling **`gse_iec_decode_done`**. |

## *Data Decode Function*

The function **`ms_asn1_to_local`** may be called to convert data from the ASN.1 representation to the "local" representation (i.e., the format expected by the "C" compiler). Information from the **`dataEntries`** element of the **`GSE_IEC_HDR`** structure may be used as the **`asn1Ptr`** and **`asn1Len`** arguments to this function.

## ms_asn1_to_local

**Usage:** This function converts data from the ASN.1 representation to the "local" representation (i.e. the format expected by the "C" compiler). This function may safely be called from multiple threads simultaneously.

**Function Prototype:**

```
ST_RET ms_asn1_to_local (RUNTIME_TYPE *runtimeTypeHead,
                         ST_INT numRuntimeTypes,
                         ST_UCHAR *asn1Ptr,
                         ST_INT asn1Len,
                         ST_CHAR *localData);
```

**Parameters:**

| | |
|---|---|
| `runtimeTypeHead` | Pointer to array of Runtime Type structures. |
| `numRuntimeTypes` | Number of Runtime Type structures in array. |
| `asn1Ptr` | Pointer to the ASN1 encoding of this data value. |
| `asn1Len` | Length of the ASN1 encoding of this data value. |
| `localData` | Pointer to local data. The decoded results are placed here. |

| | | | |
|---|---|---|---|
| **Return Value:** | `ST_RET` | `SD_SUCCESS` | No Error |
| | | `!= SD_SUCCESS` | Error |

## *Decode Done Function*

### gse_iec_decode_done

**Usage:** This function frees up all resources for the received IEC GOOSE message. This function should not be called until the message has been completely processed and the resources are no longer needed.

**Function Prototype:** `ST_RET gse_iec_decode_done (GSE_IEC_HDR *hdr);`

**Parameters:**

| | |
|---|---|
| `hdr` | A pointer to the structure allocated by a previous call to the **`gse_iec_hdr_decode`** function. This pointer must be freed when it is no longer in use, by calling **`gse_iec_decode_done`**. |

| | | | |
|---|---|---|---|
| **Return Value:** | `ST_RET` | `SD_SUCCESS` | No Error |
| | | `!= SD_SUCCESS` | Error |

## *IEC GOOSE Encode Data Structures*

```
typedef struct
  {
  ST_CHAR *dataRef;
  ST_INT elementId;
  struct runtime_type *runtimeTypeHead;    /* Array of Runtime Types           */
  ST_INT numRuntimeTypes;                  /* # of Runtime Types in array      */
  ST_CHAR *dataBuf;                        /* ptr to local data                */
  ST_VOID *userInfo;                       /* To store anything user wants.    */
                                           /* GSE code does not use it.        */
  } GSE_IEC_DATA_ENTRY;

typedef struct
  {
  ST_CHAR *gcRef;
  ST_UINT32 timeToLive;
  ST_CHAR *dataSetRef;
  ST_CHAR *appID;
  MMS_UTC_TIME utcTime;
  ST_UINT32 stNum;
  ST_UINT32 sqNum;
  ST_BOOLEAN test;
  ST_INT confRev;
  ST_BOOLEAN needsCommissioning;
  ST_INT8 sendMode;
  ST_INT numDataEntries;
  GSE_IEC_DATA_ENTRY *dataEntries;   /* array of data entry structs       */
  } GSE_IEC_CTRL;
```

## *IEC GOOSE Encode Functions*

### gse_iec_control_create

**Usage:** This function creates a IEC GOOSE control block and initializes the **gcRef**, **dataSetRef**, **appId**, **numDataEntries**, and **sendMode** elements. It also allocates an array of "data entry" control structures for the number of data entries requested (i.e. numDataEntries). It also stores a pointer to the "data entry" array in the **dataEntries** element of **GSE_IEC_CTRL**.

**Function Prototype:**

```
GSE_IEC_CTRL *gse_iec_control_create (ST_CHAR *gcRef,
                                      ST_CHAR *dataSetRef,
                                      ST_CHAR *appId,
                                      ST_INT numDataEntry,
                                      ST_UINT8 sendMode);
```

**Parameters:**

`gcRef` String to encode as gcRef in the IEC GOOSE message.

`dataSetRef` String to encode as dataSetRef in the IEC GOOSE message.

`appId` String to encode as appId in the IEC GOOSE message.

`numDataEntries` Number of Data Entries to be included in the IEC GOOSE message.

`sendMode` Sets the mode to be used for sending IEC GOOSE:

```
#define IEC_GOOSE_SEND_MODE_DATA_REF            0
#define IEC_GOOSE_SEND_MODE_ELEMENT_OFFSET      1
#define IEC_GOOSE_SEND_MODE_ALL                 2
```

**Return Value:**

`(GSE_IEC_CTRL *)` `!=NULL` Pointer to control block created.

`NULL` Control block could NOT be created.

**Comments:** The **dataEntries** array in **GSE_IEC_CTRL** is NOT initialized. The function **gse_iec_data_init** must be called for each element of the `dataEntries` array to initialize each data entry.

## gse_iec_control_destroy

**Usage:** This function destroys the resources reserved through the **`gse_iec_control_create`** function.

**Function Prototype:**

```
ST_RET gse_iec_control_destroy (GSE_IEC_CTRL *gptr);
```

**Parameters:**

| | |
|---|---|
| `gptr` | Pointer to IEC GOOSE control structure returned by **`gse_iec_control_create`**. |

| | | |
|---|---|---|
| **Return Value:** | `SD_SUCCESS` | Success |
| | `!= SD_SUCCESS` | Error |

## gse_iec_data_init

**Usage:** This function initializes a Data Entry in an existing IEC GOOSE control block.

**Function Prototype:**

```
ST_RET gse_iec_data_init  (GSE_IEC_CTRL *ctrl,
                           ST_INT index,
                           ST_CHAR *dataRef,
                           ST_INT elementId,
                           struct runtime_type *runtimeTypeHead,
                           ST_INT numRuntimeTypes);
```

**Parameters:**

`ctrl` Pointer to the IEC GOOSE control structure containing the Data Entry. This must point to a control structure created by **`gse_iec_control_create`**.

`index` Index into array of Data Entry control structures. This value must be less than **`numDataEntries`** passed to **`gse_iec_control_create`**.

`dataRef` This is the data reference used to encode in the GOOSE message for this data value (NULL if GOOSE message does not contain dataRefs).

`elementId` This is the Element ID to encode in the GOOSE message for this data value. (-1 if GOOSE message does not contain any **`elementIds`**).

`runtimeTypeHead` Pointer to array of Runtime types describing this dataEntry.

`numRuntimeTypes` Number of Runtime Types in array.

| | | | |
|---|---|---|---|
| **Return Value:** | `ST_RET` | `SD_SUCCESS` | Data entry initialized successfully. |
| | | `!= SD_SUCCESS` | Error code. |

**Comments:** If **`sendMode = IEC_GOOSE_SEND_MODE_DATA_REF`** (passed to **`gse_iec_control_create`**), then a valid dataRef must be passed to this function, and elementId will be ignored. If **`sendMode = IEC_GOOSE_SEND_MODE_ALL`** or **`IEC_GOOSE_SEND_MODE_ELEMENT_OFFSET`**, then a valid **`elementId`** must be passed to this function, and **`dataRef`** will be ignored.

It is the user's responsibility to put data entries in the right order. For example, if elements 3, 5, and 7 are to be sent, the user should call this function once with index = 0 and elementId = 3, then call again with index = 1 and elementId = 5, and again with index = 2 and elementId = 7.

## gse_iec_data_update

**Usage:** This function may be used to update the data stored for a single data entry in the IEC GOOSE control structure. Repeated calls are allowed in order to change multiple Data Entries, or to change a single data entry many times. This mechanism allows the Application to determine which values have changed and to update those values only.

Before calling this function, **`gse_iec_data_init`** must be called for this Data Entry.

**Function Prototype:**

```
ST_RET gse_iec_data_update (GSE_IEC_CTRL *ctrl,
                            ST_INT index,
                            ST_VOID *dataPtr);
```

**Parameters:**

| | |
|---|---|
| `ctrl` | A pointer to a structure containing IEC GOOSE control information. This must point to a control structure created by **`gse_iec_control_create`**. |
| `index` | Index into array of Data Entries. |
| `dataPtr` | Pointer to “local data” (i.e., normal “C” data) to be saved for this Data Entry. |

| | | | |
|---|---|---|---|
| **Return Value:** | `ST_RET` | `SD_SUCCESS` | Data entry updated successfully. |
| | | `<>SD_SUCCESS` | Error code. |

**Comments:** This function does not convert the data to ASN.1 (that is done by **`gse_iec_encode`**). Therefore, it may be called many times to update the data as it changes, with very little overhead. The size of the data to be saved is determined from the **`runtime_type`** passed to **`gse_iec_data_init`**. Therefore, it is critical that the **`runtime_type`** must correctly define the data format.

## *Encoding a IEC Goose*

## gse_iec_encode

**Usage:** This function encodes an IEC GOOSE based upon a previously created and initialized control block.

**Function Prototype:**

```
ST_UCHAR *gse_iec_encode (GSE_IEC_CTRL *ctrl, ST_CHAR *EncBuf,
                          ST_INT EncBufLen,
                          ST_INT *pEncPduLen);
```

**Parameters:**

| | |
|---|---|
| `ctrl` | A pointer to a structure containing IEC GOOSE control information. This must point to a control structure created by **`gse_iec_control_create`**. |
| `EncBuf` | A pointer to a buffer into which the GOOSE message will be encoded. |
| `EncBufLen` | The size of the encode buffer. |
| `pEncPduLen` | Pointer to length of ASN.1 encoded IEC GOOSE PDU. This is an "output" parameter. The function saves the encoded length at this address. |

| | | |
|---|---|---|
| **Return Value:**`(ST_UCHAR*)` | `NULL` | Encode error |
| | `!= NULL` | Pointer to encoded GOOSE PDU. |

**Comments:** If the data cannot be encoded according to the **`RUNTIME_TYPE`** passed to **`gse_iec_data_init`**, this function will fail.

## gse_iec_send

**Usage:** This function sends an IEC GOOSE message.

**Function Prototype:**

```
ST_RET gse_iec_send (GSE_IEC_CTRL *ctrl,
                     ST_UCHAR *dstMac,
                     ST_CHAR  *EncPduPtr,
                     ST_INT EncPduLen);
```

**Parameters:**

| | |
|---|---|
| `ctrl` | A pointer to a structure containing IEC GOOSE control information. This must point to a control structure created by **`gse_iec_control_create`**. |
| `dstMac` | A pointer to the MAC address information to which the PDU is to be sent. This must be a valid multicast MAC address. |
| `EncPduPtr` | A pointer to the ASN.1 encoded IEC GOOSE message to send. This should be the pointer returned from **`gse_iec_encode`**. |
| `EncPduLen` | The length of the ASN.1 encoded IEC GOOSE message to send. This should be the length returned from **`gse_iec_encode`**. |

| | | | |
|---|---|---|---|
| **Return Value:** | `ST_RET` | `SD_SUCCESS` | IEC GOOSE sent successfully. |
| | | `!= SD_SUCCESS` | Error code. |

# IEC-61850 GSSE (formerly UCA GOOSE)

The IEC-61850 GSSE message is implemented as a MMS InformationReport sent over the Connectionless OSI stack to multiple destinations using the multicast capability of the Ethernet MAC layer. The data to be sent in the InformationReport, as well as the timing is defined in IEC-61850. This section describes how to send and receive GSSE messages using the functions of MMS-EASE *Lite*.

## *Initialization*

The function **`clnp_read_thread_start`** may be called to start a separate thread for receiving CLNP packets and also a separate thread for decoding received GSSE packets. If this function is successful there will be three threads running when it completes: the "main" thread, the "CLNP Read" thread, and the "GSSE Read" thread.

## *Receiving GSSE Messages*

The sub-network interface function **`clnp_snet_set_multicast_filter`** must be called to enable the reception of multicast GSSE packets. It must be called during initialization, after calling **`mvl_start_acse`**. If the Ethernet driver is already set to promiscuous mode, this function does not need to be written or called.

When a multicast packet (not destined for "ALL-ES" or "ALL-IS") is received, it is assumed to be a GSSE message. It is passed to the Connectionless OSI stack. The stack decodes the packet, but does not validate any of the addressing information (i.e., PSEL, SSEL, TSEL, NSAP, and MAC). When decoding is complete, the user function **`u_mmsl_goose_received`** is called.

If **`clnp_read_thread_start`** has been called successfully, the following processing of received packets takes place:

1. The "CLNP Read" thread repeatedly calls the standard function **`clnp_read`** to receive packets. It determines if each packet is a "normal" packet or a GSSE packet and places it on the appropriate linked list and sets an appropriate event using the GLBSEM subsystem (system specific semaphore and thread handling functions – see **glbsem.c**).
2. The "main" thread waits for an event, and then calls **`mvl_comm_serve`** to process the event. However, the Transport layer of the OSI stack now calls **`clnp_read_main`** (instead of **`clnp_read`**) which gets packets from the "main" linked list.
3. The "GSSE Read" thread waits for the "GSSE Read" event, and then calls **`clpp_event`** to process the event. However, the Transport layer of the OSI stack now calls **`clnp_read_goose`** (instead of **`clnp_read`**) which gets packets from the "GSSE" linked list.
4. If the GSSE packet is successfully decoded, the user callback function **`u_mmsl_goose_received`** is called. This function is passed the decoded GSSE information, which is valid only during the function call (the user must copy the data if required for later processing).
5. To allow high priority processing of GSSE messages, the developer may choose to increase the priority of the "GSSE Read" thread, if this is possible on the target operating system. This thread would then interrupt standard MMS message processing when a GSSE packet is received.

If **`clnp_read_thread_start`** is NOT called, incoming packets are processed in the order they are received using a single "main" thread.

1. The "main" thread waits for an event, and then calls **`mvl_comm_serve`** to process the event. However, the Transport layer of the OSI stack now calls **`clnp_read`**, which reads packets from the Ethernet driver (not from a linked list).
2. If a GSSE Packet is received and successfully decoded, the user callback function **`u_mmsl_goose_received`** is called. This function is passed the decoded GSSE information, which is valid only during the function call (the user must copy the data if required for later processing).

## *Sending GSSE Messages*

To send a GSSE message, simply call the function **`mmsl_send_goose`**. This can be done from the "main" thread or from any other thread. If this is done from another thread, however, the "stack" library must be compiled with **`S_MT_SUPPORT`** defined. This is necessary to make sure the eventual call to **`clnp_write`** is threadsafe.

## *Porting Issues*

Complete the instructions in each of the following sections for successful porting.

### GSSE Source Code

The source file **goose.c** in the directory **\mmslite\uca\goose**, must be included in the "stack" makefile (i.e., **ositpxe.mak**, **ositp4e.mak**, etc.).

### GSSE Header File

The header file **goose.h** in the directory **\mmslite\inc**, must be included in any user source modules that will be performing GSSE processing.

### GLBSEM subsystem

If the multi-threading features are to be used, the source file **glbsem.c** must be ported to the target operating system.

### Makefile changes

The following changes must be made to makefiles (or DSP files for Windows):

| | |
|---|---|
| mem.mak - | Add "`-D S_MT_SUPPORT`". |
| slog.mak - | Add "`-D S_MT_SUPPORT`". |
| util.mak - | Add "`-D S_MT_SUPPORT`". Add "`glbsem.c`". |
| ositpxe.mak or ositp4e.mak - | Add "`clnp_thr.c`". |
| uositpxe.mak or uositp4e.mak - | Add "`-D CLACSE`"<br>Add "`–D MVL_GOOSE_SUPPORT`"<br>Add "`–D S_MT_SUPPORT`" |

**Application changes**

The source file **uca_srvr.c** may be examined to see how to use the new features. Note the following items:

- **clnp_snet_set_multicast_filter** must be called AFTER **mvl_start_acse** to receive multicast packets.
- The user callback function **u_mmsl_goose_received** must be written to examine the GSSE packets received. A very simple example is included.
- **mmsl_send_goose** must be called to send a GSSE packet.

If a separate thread for receiving GSSE packets is desired:

- **clnp_read_thread_start** must be called AFTER **mvl_start_acse**.

If a separate thread for sending GSSE packets is desired:

- An example of a GSSE transmit thread **goose_tx_thread** is included.

## GSSE Data Structures

```
typedef struct
  {
  ST_UCHAR   loc_mac [CLNP_MAX_LEN_MAC]; /* local MAC address     */
  ST_UCHAR   rem_mac [CLNP_MAX_LEN_MAC]; /* remote MAC address    */
  ST_UINT16  lpdu_len;       /* Length of LPDU.                   */
  ST_UCHAR  *lpdu;           /* Pointer to LPDU buffer to send.  */
  }SN_UNITDATA;

typedef struct tagAUDT_APDU
  {
  /* The following entries passed to peer in AUDT-apdu.           */
  /* ACSE sets and checks "protocol-version".  Must be "version1".*/
  ST_BOOLEAN      ASO_context_name_pres;
  MMS_OBJ_ID      ASO_context_name;
  AE_TITLE  called_ae_title;
  AE_TITLE  calling_ae_title;
  BUFFER    user_info;       /* User must encode/decode    */

  /* The following entries passed to or received from presentation.*/
  PRES_ADDR calling_paddr;
  PRES_ADDR called_paddr;

  /* User doesn't need to set loc_mac before calling a_unit_data_req*/
  /* Decode process fills in loc_mac before calling u_a_unit_data_ind*/
  ST_UCHAR    loc_mac  [CLNP_MAX_LEN_MAC];      /* Local MAC addr */
  ST_BOOLEAN rem_mac_valid;         /* SD_TRUE if MAC addr valid  */
  ST_UCHAR    rem_mac  [CLNP_MAX_LEN_MAC];      /* Remote MAC addr*/
  } AUDT_APDU;
```

```
typedef struct
  {
/* GSSE Stack addressing information */
  AUDT_APDU audtApdu;

/* GSSE MMS values */
  ST_CHAR  SendingIED[66];
  MMS_BTOD  t;
  ST_UINT32 SqNum;
  ST_UINT32 StNum;
  ST_UINT32 HoldTim;
  ST_UINT32 BackTim;
  ST_UINT16 PhsID;

  ST_INT num_dna_bits;
  ST_UCHAR  DNA[GOOSE_MAX_NUM_DNA_BITS/8];

  ST_INT num_usr_bits;
  ST_UCHAR  UserSt[GOOSE_MAX_NUM_USR_BITS/8];
  } GOOSE_INFO;
```

**Fields**

`SendingIED` The SendingIED uniquely names the device reporting to GSSE. A given reporting IED may handle several devices.

`t` The time in milliseconds of the substation event. This time changes each time a substation event occurs.

`SqNum` This sequence number is incremented by one each time a message is sent. It rolls over after the max count is reached.

`StNum` This state number is incremented by one each time an IED sends information that is new.

`HoldTim` Hold time. The time that a particular message (status) is held before it is canceled. Cancellation, depending on the status reported, may result in an automatic reset of the conditions (i.e., a BFI timer is canceled or the removal of a “block” condition). In order for the status conditions within the message to remain valid, a repeat of the message must be received before the hold time expires. The hold time is incremented each time the message is sent and may follow geometric progression (e.g., 10, 20, 30, 40, 50 ....... up to a maximum of one minute). The timers and progressions are parameterized. The progression timers reset when a new GSSE is created.

`BackTim` This is the microsecond offset beteen the time **`t`** and the actual time of the substation event.

`PhsID` This integer value defines which Phases are involved. GOMSFE defines these as follows.

| **Value** | **Name** |
|---|---|
| 0 | *None* |
| 1 | *Phase* |
| 2 | *Phase B* |
| 3 | *Phase C* |
| 4 | *Ground Only* |
| 5 | *A to Ground* |
| 6 | *B to Ground* |
| 7 | *C to Ground* |
| 8 | AB |
| 10 | *CA* |

| | |
|---|---|
| 11 | *AB to Ground* |
| 12 | *BC to Ground* |
| 13 | *CA to Ground* |
| 14 | *ABC* |
| 15 | *ABC to Ground* |

`num_dna_bits` This is the number **n** of bits to send in the DNA bitstring numbered 0 to n.

`DNA` DNA is a single message that conveys all genetically required protection scheme information regarding an individual IED. This message uniquely reports the status of the described elements resident in the ***transmitting*** IED to its peers per the enrollment list. Please refer to the GOOSE Message Section of GOMSFE for values of bit pairs.

`num_usr_bits` This is the number **n** of bits to send in the User Status bitstring numbered 0 to n.

`UserSt` These bit pairs are user defined and are available for statuses not covered in the DNA. Their meaning is assumed to be known and understood between applications exchanging them.

## *GSSE Handling Functions*

## mvl_init_audt_addr

**Usage:** This function is used to get connectionless addressing information for later use when sending GSSE messages. The calling and called AE Title and P-Address information are copied to the **AUDT_APDU** structure from local and remote DIB entry information. The MMS AP context of {1,0,9506,2,3} is assigned to the **ASO_context_name** of the **AUDT_APDU**.

**Function Prototype:**

```
ST_RET mvl_init_audt_addr (AUDT_APDU *audt,
                           ST_CHAR *localArName,
                           ST_CHAR *remoteArName)
```

**Parameters:**

| | |
|---|---|
| audt | This output parameter contains **AUDT_APDU** information useful for copying to the **audtApdu** field of the **GOOSE_INFO** prior to sending a GSSE message. |
| locArName | This is the ASCII character string of a local DIB entry used for the *calling* **AUDT_APDU** information. |
| remARName | This is the ASCII character string of a remote DIB entry sed for the *called* **AUDT_APDU** information. |

| **Return Value:** | | | |
|---|---|---|---|
| | ST_INT | SD_SUCCESS | Initialization was OK. |
| | | MVLE_LOCAL_NAME | **locArName** not found in local DIB. |
| | | MVLE_REMOTE_NAME | **remArName** not found in local DIB. |

## mmsl_send_goose

**Usage:** This function is called to broadcast a GSSE message. It may be called from any thread, but due to the critical nature of PACT, it is recommended to call this function from a separate thread as soon as the substation event occurs.

**Function Prototype:** `ST_RET mmsl_send_goose (GOOSE_INFO *gi);`

**Parameters:**

`gi` A pointer to a **`GOOSE_INFO`** structure that is the source information for the GSSE message. The **`GOOSE_INFO`** structure is defined in **goose.h**.

| | | | |
|---|---|---|---|
| **Return Value:** | `ST_RET` | `SD_SUCCESS` | The GSSE message was sent OK. |
| | | `<>0` | Error code. |

**Comments:** An identical function named **`gse_uca_write`** may also be used (one of these functions may be implemented as a macro).

## u_mmsl_goose_received

**Usage:** This user function is called when a GSSE message is received. This function must be written by the user to examine and/or process the information received in the GSSE message.

**Function Prototype:** `ST_VOID u_mmsl_goose_received (GOOSE_INFO *goose_info);`

**Parameters:**

`goose_info` A pointer to a **`GOOSE_INFO`** structure that contains the GSSE information received from the network.

**Return Value:** `ST_VOID`

## clnp_read_thread_start

**Usage:** This function is called to start up separate threads for receiving network transactions. After calling this function, all GSSE messages received from the network will immediately be sent to the **`u_mmsl_goose_received`** function. OSI and TCP/IP traffic will continue to be processed by calling **`mvl_comm_serve`**.

**Function Prototype:** `ST_RET clnp_read_thread_start (ST_VOID);`

**Parameters:** NONE

| | | | |
|---|---|---|---|
| **Return Value:** | `ST_RET` | `SD_SUCCESS` | Threads started OK. |
| | | `<>0` | Error code. |

## *Alternate GSSE Reception Mode*

Applications that need to receive GSSE messages, but do not need to support the 7-layer OSI stack, may be able to use the following method to receive the GSSE messages. This method allows the programmer more direct control of the reception and decoding of subnetwork packets containing GSSE messages.

## *Receiving GSSE Messages*

1. The sub-network interface function **`clnp_snet_set_multicast_filter`** must be called to specify a list of multicast MAC addresses to be accepted. This function enables the reception of multicast packets addressed to a set of multicast MAC addresses selected by the user.
2. The function **`clnp_snet_read`** must be called periodically to receive packets from the network.
3. The destination MAC address and the LLC LSAP in the packet may be examined to determine if the packet contains a GSSE message. The user may choose to process any packets of interest. The packets may be stored for later processing (e.g., on a queue or a linked list) or they may be processed immediately.
4. If a received packet contains a GSSE message, the function **`gse_uca_decode`** may be called to decode the packet. This function places the decoded information in a **`GOOSE_INFO`** structure to be examined by the user.
5. If desired, the user code may perform additional processing on the GSSE data contained in the **`GOOSE_INFO`** structure.
6. The function **`clnp_snet_free`** must be called to free the Ethernet packet after all processing is complete.

## *Multithreading - Using Separate Threads to Receive and to Decode GSSE Packets*

A separate thread may be spawned to periodically call **`clnp_snet_read`** to receive incoming packets and put them on queues. Any appropriate “thread-safe” queuing mechanism may be used (e.g., the SISCO linked list functions), and any number of queues may be maintained. For example, one queue may contain only GSSE packets, while another queue contains all other packets. Another thread may be spawned to get GSSE packets from the queues and call **`gse_uca_decode`** to decode them. Thread priorities may be adjusted, if desired, so that GSSE packets are processed at a higher priority than other packets.

## *Additional Functions for Alternate GSSE Reception Mode*

### gse_uca_decode

**Usage:** This function decodes the received GSSE packet from the subnetwork layer up through the application layer.

**Function Prototype:**
```
ST_RET gse_uca_decode (SN_UNITDATA *sn_udt,
                       GOOSE_INFO *goose_info);
```

**Parameters:**

| | |
|---|---|
| `sn_udt` | This is a packet received from the subnetwork interface. |
| `goose_info` | This is a pointer to a **`GOOSE_INFO`** structure to contain the result of the decode. |

| | | | |
|---|---|---|---|
| **Return Value:** | `ST_RET` | `SD_SUCCESS` | The GSSE message was decoded correctly. |
| | | `<>0` | Error code. |

## Appendix A

# Subset Creation

Since MMS-EASE *Lite* 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.

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.

## Appendix B

# 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.
- Controllable options:
    - File Name
    - File Size
    - Wipe Bar
    - File Wrap
    - Message Header
    - Append/Overwrite on open
    - Hard Flush
    - Setbar Control

# Memory Logging

Below is a feature 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. Additionally, 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;
```

**<u>Fields</u>**:

`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. |

`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 | */ |

# IPC LOGGING

The IPC Logging option allows sending log messages over a TCP connection. The module **mmslite\src\slogipc.**c must be linked to the application to enable this functionality.

## *IPC Logging in Application*

An application can enable IPC logging by setting the following flag in the SISCO's global log control structure:

```
sLogCtrl->logCtrl |= LOG_IPC_EN;
```

This IPC logging flag can be enabled/disabled multiple times while the application is running. Although not required the application can call the initialization function **`slogIpcInit (sLogCtrl)`** when it is setting first time the **`LOG_IPC_EN`** flag. This will allow the IPC logging system to send application identification message to a Client as soon as socket connection is established, before any log messages are sent.
The default listening port designated by SISCO for an application is IPC_LOG_BASE_PORT (55147) defined in the **slog.h**.

Note that MMS-EASE libraries have only error log mask turned ON. If application wants to log various levels of MMS communication/processing it needs to turn proper masks ON.

The application can change the default IPC logging parameters by modifying fields in the **sLogCtrl->ipc (IPC_LOG_CTRL)** structure:

port Base port number where application will listen for socket connections from Client applications such as Hyper Terminal or Telnet. Default is **IPC_LOG_BASE_PORT** (55147).

portCnt Number of listening ports starting with base port, that are available to multiple instances of the application. Default is 1.

portUsed This is the listen port actually used by given instance of an application. Set in the **slogIpcInit** function.

maxConns Maximum number of socket connections that can be accepted for IPC logging. The default is **IPC_LOG_MAX_CONNECTIONS** (10).

maxQueCnt Maximum number of log messages that can be queued on any one connection. The default is **IPC_LOG_MAX_QUEUE_CNT** (100).

appId This is pointer to a NULL terminated string identifying the application. The buffer holding this information must be persistent while the program is running. There is no size limit on the buffer. The application identification string is sent to a Client in the first message after socket connection has been established. The default is NULL pointer.

# 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**ing. 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;
fc.maxSize = 1000000;
fc.fileName = "mms.log";
fc.ctrl = FIL_CTRL_WIPE_EN | FIL_CTRL_WRAP_EN | FIL_CTRL_MSG_HDR_EN;
```

## *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`**. |
| `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`

## 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`**.

`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`

## 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.

`newfile` This is a pointer to a string containing the new file name.

**Return Value:** `ST_VOID`

## 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**.

**Return Value:** `ST_VOID`

## 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`**.

**Return Value:** `ST_INT` 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 Pointer Global Variable:**

```
extern ST_VOID (*slog_dyn_log_fun) (LOG_CTRL *lc,
                                    ST_INT logType,
                                    ST_CHAR *sourceFile,
                                    ST_INT lineNum,
                                    ST_INT bufLen,
                                    ST_CHAR *buf);
```

**Parameters**:

`lc` This is a pointer to the log control structure of type **`LOG_CTRL`**.

`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.

## slog_dyn_log_fun (cont'd)

**Parameters**:

| | |
|---|---|
| `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`

**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 Pointer Global Variable:** `extern ST_VOID (*slog_service_fun) (ST_VOID);`

**Parameters:** NONE

**Return Value:** `ST_VOID`

# Enhanced Slogging Features

SISCO provides library functions and new macros to help application developers implement logging. The following sample demonstrates how to implement logging.

**In an application header file (e.g., myapp.h):**

```
  /* logging masks */
  #define MYLOG_ERR        0x00000001
  #define MYLOG_FLOW       0x00000002
  #define MYLOG_DATA       0x00000004

  extern ST_UINT my_debug_sel;

  extern SD_CONST ST_CHAR *SD_CONST _mylog_err_logstr;
  extern SD_CONST ST_CHAR *SD_CONST _mylog_flow_logstr;

  /*  error log macros /
  #define MY_LOG_ERR0(a)    SLOG_0 (my_debug_sel & MYLOG_ERR,_mylog_err_logstr, a)
  #define MY_LOG_ERR1(a, b) SLOG_1 (my_debug_sel & MYLOG_ERR,_mylog_err_logstr, a, b)
  /* Create new macros MY_LOG_ERR2, etc.,                  */
  /* for each additional argument passed to the log macro. */

  /*  error log continuation macros (do not include log message header) */
  #define MY_LOG_ERRC0(a)    SLOGC_0 (my_debug_sel & MYLOG_ERR,_mylog_err_logstr, a)
  #define MY_LOG_ERRC1(a, b) SLOGC_1 (my_debug_sel & MYLOG_ERR,_mylog_err_logstr, a, b)

  /* program flow log macros */
  #define MY_LOG_FLOW0(a)   SLOG_0 (my_debug_sel & MYLOG_FLOW,_mylog_flow_logstr, a)

  /* hex logging */
  #define MY_LOG_DATA(num, ptr)   SLOGH (my_debug_sel & MYLOG_DATA, num , ptr)
```

**In an application’s C or C++ file:**

```
  #include “glbtypes.h”    /* SISCO’s file */
  #include “slog.h”        /* SISCO’s file */
  #include “myapp.h”

  SD_CONST static ST_CHAR *SD_CONST thisFileName = __FILE__;

  ST_UINT my_debug_sel = MYLOG_ERR;
/* log errors, other masks maybe set during program execution  */

  SD_CONST ST_CHAR *SD_CONST _mylog_err_logstr =  “MYLOG_ERR”;
  SD_CONST ST_CHAR *SD_CONST _mylog_flow_logstr = “MYLOG_FLOW”;



  /* sample of application logging */
  if  (type == expected_type)
    {
    MY_LOG_FLOW1 ("Received message type= %d", type);
    MY_LOG_DATA (num_bytes, data_ptr);
    }
  else
    MY_LOG_ERR1 ("Unexpected message received type= %d", type);
```

SISCO logging functions can be accessed directly but the SLOG macros are a more convenient and simpler way of writing logging code.

# MMS-EASE *Lite* Log Levels

The amount of logging produced by MMS-EASE *Lite* is controlled by setting global MMS-EASE variables. These variables hold log control bits for enabling and disabling the various levels of logging as shown below.

## mms_debug_sel

```
extern ST_ULONG mms_debug_sel;
```

| CONSTANT | BIT ASSIGNMENTS | ENABLE LOGGING OF... |
|---|---|---|
| MMS_LOG_DEC | 0x00000001L | MMS decoding process |
| MMS_LOG_ENC | 0x00000002L | MMS encoding process |
| MMS_LOG_ERR | 0x00010000L | Abnormal errors |
| MMS_LOG_NERR | 0x00020000L | Normal errors |
| MMS_LOG_RT | 0x00010000L | All RunTime Type transactions |
| MMS_LOG_RTAA | 0x00020000L | All RunTime Type AlternateAccess transactions |
| MMS_LOG_AA | 0x00040000L | All Alternate Access transactions |

The following defines are user-reserved. These are not used by MMS-EASE

| | | |
|---|---|---|
| MMS_LOG_USR_IND | 0x00000100L | User Indications |
| MMS_LOG_USR_CONF | 0x00000200L | User Confirmations |

By default, **`mms_debug_sel`** is set to **`MMS_LOG_ERR`**.

## asn1_debug_sel

```
extern ST_UINT asn1_debug_sel;
```

| CONSTANT | BIT ASSIGNMENTS | ENABLE LOGGING OF... |
|---|---|---|
| ASN1_LOG_DEC | 0x0001 | ASN.1 decode process |
| ASN1_LOG_ENC | 0x0002 | ASN.1 encode process |
| ASN1_LOG_ERR | 0x0004 | Abnormal ASN.1 errors |
| ASN1_LOG_NERR | 0x0008 | Normal ASN.1 errors |

By default, **`asn1_debug_sel`** is set to **`ASN1_LOG_ERR`**.

## list_debug_sel

```
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.

# chk_debug_en

```
extern ST_UINT chk_debug_en;
```

| CONSTANT | BIT ASSIGNMENTS | ENABLE LOGGING OF... |
|---|---|---|
| MEM_LOG_ERR | 0x0001 | Abnormal memory errors |
| MEM_LOG_MALLOC | 0x0002 | **chk_malloc** calls |
| MEM_LOG_CALLOC | 0x0004 | **chk_calloc** calls |
| MEM_LOG_REALLOC | 0x0008 | **chk_realloc** calls |
| MEM_LOG_FREE | 0x0010 | **chk_free** calls |

By default, **chk_debug_en** is set to **MEM_LOG_ERR**.

# mvl_debug_sel

The global variable **mvl_debug_sel** may be used to control the logging of the MVL layer.

```
extern ST_UINT mvl_debug_sel;
```

The following values may be used to set the global variable **mvl_debug_sel** to enable different types of logging in the MVL layer.

| CONSTANT | BIT ASSIGNMENTS | ENABLE LOGGING OF... |
|---|---|---|
| MVLLOG_ERR | 0x00000001 | MVL Critical Errors |
| MVLLOG_NERR | 0x00000002 | MVL Normal Errors |
| MVLLOG_ACSE | 0x00000040 | MVL ACSE Encoding/Decoding |
| MVLLOG_ACSEDATA | 0x00000080 | MVL ACSE Encoding/Decoding HEX data |
| MVLULOG_FLOW | 0x00000200 | MVL UCA Report Flow |

# acse_debug_sel

The global variable **acse_debug_sel** may be used to control the logging of the ACSE, Presentation (COPP), and Session (COSP) layers of the OSI stack (the Presentation and Session settings are included here simply to avoid extra unnecessary global variables).

```
extern ST_UINT acse_debug_sel;
```

The following values may be used to set the global variable **acse_debug_sel** to enable different types of logging in the ACSE, Presentation (COPP), and Session (COSP) layers.

| CONSTANT | BIT ASSIGNMENTS | ENABLE LOGGING OF... |
|---|---|---|
| ACSE_LOG_ERR | 0x00000001 | ACSE Errors |
| ACSE_LOG_ENC | 0x00000002 | ACSE Encoding |
| ACSE_LOG_DEC | 0x00000004 | ACSE Decoding |
| ACSE_LOG_DIB | 0x00000008 | ACSE DIB (i.e. network addressing) |
| COPP_LOG_ERR | 0x00000100L | COPP Errors |
| COPP_LOG_DEC | 0x00001000L | COPP Decoding |
| COPP_LOG_DEC_HEX | 0x00002000L | COPP Decoding Hex |
| COPP_LOG_ENC | 0x00004000L | COPP Encoding |
| COPP_LOG_ENC_HEX | 0x00008000L | COPP Encoding Hex |
| COSP_LOG_ERR | 0x00010000L | COSP Errors |

| | | |
|---|---|---|
| COSP_LOG_DEC | 0x00100000L | COSP Decoding |
| COSP_LOG_DEC_HEX | 0x00200000L | COSP Decoding Hex |
| COSP_LOG_ENC | 0x00400000L | COSP Encoding |
| COSP_LOG_ENC_HEX | 0x00800000L | COSP Encoding Hex |

Another global variable, **sLogCtrl**, is used by all SISCO components to set general logging parameters such as the log file name. It is defined as follows:

```
LOG_CTRL *sLogCtrl;
```

By default, **sLogCtrl** points to a structure with no logging enabled. This structure must be modified before any logging will begin.

# tp4_debug_sel

To assist the user in diagnosing communication and other TP4 API-related problems, the optional SISCO Logging Facility (slog) may be used. Refer to separate documentation of the SISCO Logging Facility for details. To control TP4 API logging, a global variable, **tp4_debug_sel**, is provided. It is used to select the amount and nature of logging produced by TP4 API. It may be set to one of the following values:

| CONSTANT | BIT ASSIGNMENTS | ENABLE LOGGING OF... |
|---|---|---|
| TP4_LOG_ERR | 0x00000001 | Transport Errors |
| TP4_LOG_FLOWUP | 0x00000002 | Transport Decode (incoming TPDUs) |
| TP4_LOG_FLOWDOWN | 0x00000004 | Transport Encode (outgoing TPDUs) |

If more than one type of logging is desired, the bitwise OR operator may be used as in the following example:

```
tp4_debug_sel= TP4_LOG_ERR | TP4_LOG_FLOWUP | TP4_LOG_FLOWDOWN;
```

The default setting is to log errors only as follows:

```
tp4_debug_sel = TP4_LOG_ERR;
```

Another global variable, **sLogCtrl**, is used by all SISCO components to set general logging parameters, such as log file name. It is defined as follows:

```
LOG_CTRL *sLogCtrl;
```

By default, **sLogCtrl** points to a structure with no logging enabled. This structure must be modified before any logging will begin.

# clnp_debug_sel

To assist in diagnosing communication and other CLNP API related problems, the optional SISCO Logging Facility (slog) may be used. Refer to separate documentation of the SISCO Logging Facility for details. To control CLNP API logging, a global variable, **clnp_debug_sel**, is provided. It is used to select the amount and nature of logging produced by the CLNP API. It may be set to one of the following values:

| CONSTANT | BIT ASSIGNMENTS | ENABLE LOGGING OF... |
|---|---|---|
| CLNP_LOG_ERR | 0x00000001L | CLNP Critical Errors |
| CLNP_LOG_NERR | 0x00000002L | CLNP Normal Errors |
| CLNP_LOG_REQ | 0x00000010L | CLNP Requests |

| | | |
|---|---|---|
| CLNP_LOG_IND | 0x00000020L | CLNP Indications |
| CLNP_LOG_ENC_DEC | 0x00000100L | CLNP Encode/Decode |
| CLNP_LOG_LLC_ENC_DEC | 0x00000200L | LLC Encode/Decode |
| CLSNS_LOG_REQ | 0x00001000L | CLSNS (Subnetwork) Requests |
| CLSNS_LOG_IND | 0x00002000L | CLSNS (Subnetwork) Indications |

If more than one type of logging is desired, the bitwise OR operator may be used as in the following example:

```
clnp_debug_sel = CLNP_LOG_ERR | CLNP_LOG_REQ | CLNP_LOG_IND;
```

Another global variable, **sLogCtrl**, is used by all SISCO components to set general logging parameters, such as log file name. It is defined as follows:

```
LOG_CTRL *sLogCtrl;
```

By default, **sLogCtrl** points to a structure with no logging enabled. This structure must be modified before any logging will begin.

# smp_debug_sel

The global variable **smp_debug_sel** may be used to control the logging of the SMP layer.

```
extern ST_UINT smp_debug_sel;
```

The following values may be used to set the global variable **smp_debug_sel** to enable different types of logging in the SMP layer.

| CONSTANT | BIT ASSIGNMENTS | ENABLE LOGGING OF... |
|---|---|---|
| SMP_LOG_ERR | 0x00000001L | SMP Errors |
| SMP_LOG_REQ | 0x00000010L | SMP Requests |
| SMP_LOG_IND | 0x00000020L | SMP Indications |
| SMP_LOG_HEX | 0x00000080L | Hex encoding of SMP Packets |

## Appendix C

# Linked List Manipulation

MMS-EASE provides a set of data structures and functions that allow 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 using one set of list manipulation primitives 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. |

## *Generic Link List Handling Functions*

### 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 to a pointer to the first element of the list of type **`DBL_LNK`**. |
| `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. |

Appendix D

# Memory Management Tools

There are two types of Memory Management for MMS-EASE *Lite*, Standard and Pooled. By default, Standard Memory Management is used.

## Standard Memory Mangement

MMS-EASE provides a set of memory management tools that include logging and integrity checking. To do so, replacement macros for the standard C runtime library functions **`malloc`**, **`calloc`**, **`realloc`**, and **`free`** are provided. These replacement macros are **`chk_malloc`**, **`chk_calloc`**, **`chk_realloc`**, and **`chk_free`**, respectively. These macros accept the same arguments as their counterparts from the standard C runtime library and are used internally by MMS-EASE. The macros are exposed so that MMS-EASE applications 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. 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, 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.

3. 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 with Standard Memory Management

All memory allocation in MMS-EASE Lite is done via macros (defined in **mem_chk.h**). The macros are defined such that different functions are called depending on how the source code is compiled. To use the Standard Memory Management, all source code must be compiled with **DEBUG_SISCO** defined (**SMEM_ENABLE** *NOT* defined), and then the following macros are used:

```
#define M_MALLOC(ctx,x)       x_chk_malloc  (x,  thisFileName,__LINE__)

#define M_CALLOC(ctx,x,y)     x_chk_calloc  (x,y,thisFileName,__LINE__)

#define M_REALLOC(ctx,x,y)    x_chk_realloc (x,y,thisFileName,__LINE__)

#define M_STRDUP(ctx,x)       x_chk_strdup  (x,  thisFileName,__LINE__)

#define M_FREE(ctx,x)         x_chk_free    (x,  thisFileName,__LINE__)


#define chk_malloc(x)         x_chk_malloc  (x,  thisFileName,__LINE__)

#define chk_calloc(x,y)       x_chk_calloc  (x,y,thisFileName,__LINE__)

#define chk_realloc(x,y)      x_chk_realloc (x,y,thisFileName,__LINE__)

#define chk_strdup(x)         x_chk_strdup  (x,  thisFileName,__LINE__)

#define chk_free(x)           x_chk_free    (x,  thisFileName,__LINE__)
```

Notice that the **M_MALLOC** and **chk_malloc** macros produce the same result. The **ctx** argument to **M_MALLOC** is not used. Similarly, the **M_CALLOC** and **chk_calloc** macros produce the same result, and so on.

Every module using these memory management macros must also define a static variable, **thisFilename**, and 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_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`**.

# Dynamic Memory Allocation

## *Dynamic Memory Allocation Functions*

### 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`

### 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
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`

## 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`

## 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. |

## x_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 are 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 *x_chk_malloc   (ST_UINT size,
                         ST_CHAR *file,
                         ST_INT line);
```

**Parameters**:

| | |
|---|---|
| `size` | This indicates the size in bytes of the buffer to be allocated. |
| `file` | Name of source file where this function is called. |
| `line` | Line number in source file where this function is called. |

| **Return Value:** | | | |
|---|---|---|---|
| | `ST_VOID *` | `<> null` | This is a pointer to the allocated buffer. |
| | | `= null` | The memory allocation has failed. |

## x_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 *x_chk_calloc   (ST_UINT num,
                         ST_UINT size,
                         ST_CHAR *file,
                         ST_INT line);
```

**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. |
| `file` | Name of source file where this function is called. |
| `line` | Line number in source file where this function is called. |

| **Return Value:** | | | |
|---|---|---|---|
| | `ST_VOID *` | `<> null` | This is a pointer to the allocated buffer. |
| | | `= null` | The memory allocation failed. |

## x_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 *x_chk_realloc  (ST_VOID *old,
                         ST_UINT size,
                         ST_CHAR *file,
                         ST_INT line);
```

**Parameters:**

| | |
|---|---|
| `old` | This pointer indicates the old buffer. |
| `size` | This indicates the new size of the buffer. |
| `file` | Name of source file where this function is called. |
| `line` | Line number in source file where this function is called. |

**Return Value:**

| | | |
|---|---|---|
| `ST_VOID *` | `<> null` | This is a pointer to the resized buffer. |
| | `= null` | The memory reallocation failed. |

## x_chk_free

**Usage:** This function deallocates a memory buffer allocated with **`x_chk_calloc`**, **`x_chk_malloc`**, or **`x_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 x_chk_free (ST_VOID *ptr,
                    ST_CHAR *file,
                    ST_INT line);
```

**Parameters**:

| | |
|---|---|
| `ptr` | This is a pointer to the memory buffer that is to be deallocated. |
| `file` | Name of source file where this function is called. |
| `line` | Line number in source file where this function is called. |

**Return Value:** `ST_VOID`

### 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 Pointer Global Variable:** `extern ST_VOID (*mem_chk_err) (ST_VOID);`

**Parameters:** NONE

**Return Value:** `ST_VOID`

# Pooled Memory Management Using SMEM

The SMEM Memory Manager allows the user to create pools of memory buffers. An application may then obtain memory buffers from the pools instead of using the system memory allocation functions (**`malloc`**, etc.). By avoiding use of the system memory allocation functions, **memory fragmentation** can be eliminated. SMEM also allows the user to monitor or track the usage of the pools defined. The information produced can help the user adjust the input parameters and thus configure different areas of memory to produce more desirable results.

The pools are generally created at startup. When an application calls SMEM to obtain a memory buffer, SMEM finds an available buffer in its pools, and returns a pointer to the buffer

Parameters specific to SMEM Memory Management may be configured by data entered in the **smemcfg.xml** file (an example may be found in the directory **\mmslite\mvl\usr\uca_srvr**). This file is parsed by calling the function **`smemcfgx`** (in **smemcfgx.c**) and the results are placed in various global data structures for later use.

## Compiling and Linking with Pooled Memory Management

All memory allocation in MMS-EASE Lite is done using macros (defined in **mem_chk.h**). The macros are defined such that different functions are called depending on how the source code is compiled.  To use the Pooled Memory Management, all source code must be compiled with **`DEBUG_SISCO`** *and* **`SMEM_ENABLE`** defined, and then the following macros are used:

```
#define M_MALLOC(ctx,x)    x_m_malloc  (ctx,x,  thisFileName,__LINE__)

#define M_CALLOC(ctx,x,y)  x_m_calloc  (ctx,x,y,thisFileName,__LINE__)

#define M_REALLOC(ctx,x,y) x_m_realloc (ctx,x,y,thisFileName,__LINE__)

#define M_STRDUP(ctx,x)    x_m_strdup  (ctx,x,  thisFileName,__LINE__)

#define M_FREE(ctx,x)      x_m_free    (ctx,x,  thisFileName,__LINE__)

#define chk_malloc(x)      x_m_malloc  (MSMEM_GEN,x,  thisFileName,__LINE__)

#define chk_calloc(x,y)    x_m_calloc  (MSMEM_GEN,x,y,thisFileName,__LINE__)

#define chk_realloc(x,y)   x_m_realloc (MSMEM_GEN,x,y,thisFileName,__LINE__)

#define chk_strdup(x)      x_m_strdup  (MSMEM_GEN,x,  thisFileName,__LINE__)

#define chk_free(x)        x_m_free    (MSMEM_GEN,x,  thisFileName,__LINE__)
```

Notice that the **`M_MALLOC`** and **`chk_malloc`** macros produce the same result, except that the **`chk_malloc`** macro assumes the **`ctx`** argument is always **`MSMEM_GEN`** (i.e., the General context is always used). Similarly, the **`M_CALLOC`** and **`chk_calloc`** macros produce the same result, and so on.

Every module using these memory management macros must also define a static variable, **`thisFilename`**, and include **mem_chk.h**, as follows:

```
#ifdef DEBUG_SISCO
static char *thisFileName = __FILE__;
#endif

#include "mem_chk.h"
```

# SMEM Contexts

One or more contexts may be configured in the **smemcfg.xml** file. Each context defined will contain a list of available memory pools and optional range tracking information.

A context contains a list of memory pools stored from smallest to largest. Multiple pools of the same size are allowed. Each pool is defined by a buffer size, the number of buffers, and optional parameters that specify if auto cloning can be implemented and if so, the maximum number of clones allowed

A context may also contain range limits for monitoring or tracking the usage of the defined pools. Up to a specified number of range limits may be entered. Each set of limits contains a high limit and a low limit. The specified ranges may overlap.

# SMEM Pools

Each SMEM context may contain many SMEM pools. The following parameters may be specified to create a pool: the name of the pool, the size of each buffer, the number of buffers, whether cloning is allowed, and if so, the maximum number of clones that may be created. All parameters are read from the **smemcfg.xml** file and stored in a pool control block.

If either the buffer size OR the number of buffers is omitted from the configuration of a pool, then the user function **`u_smem_get_pool_params`** is called to get the necessary information.

Selecting auto cloning for a pool of memory allows more memory pools of that size to be created if none are available. Non-availability occurs because all existing pools of the specified size are being used or no pool(s) of the specified size exist.

If “auto clone” is not configured in the configuration, then it will default to ‘No’.

If “max clones” is not configured in the configuration, then it will default to an infinite number of clones.

# “System” Memory Allocated when Creating Pools

SMEM creates a memory pool by calling **`malloc`** to obtain a large block of memory from the operating system. All the SMEM buffers are contained within this large block (i.e. **`BlockSize = NumberOfBuffers * BufferSize`**). Small amounts of additional memory are allocated for the Pool Control structure and other overhead.

# Allocating Memory with SMEM

The SMEM context is passed to the SMEM allocation function.

A search is made for the first pool in the selected context large enough to hold the requested buffer size. If none exist, the message "SMEM has no buffers large enough for size …" is logged and an attempt to create one is made. If that fails then the error message "Error: no SMEM control elements for pool size …" is logged.

If a pool larger then or equal to the requested size exists then a check is made for availability within that pool. If all the buffers in the pool are used up the message "SMEM needs more buffers of size …" is logged and a check is made for another pool of that same size. If the buffers in the second pool are all used up then the message "User did not supply enough buffers of size  …" is logged and no memory is allocated.

# Freeing Memory with SMEM

The SMEM context is passed to the SMEM free function.

A context's pool list is searched for the specified buffer to be freed. If the buffer is not found in the list, the message "`SMEM free could not find SMEM pool control for the ptr …`", and the buffer is NOT freed.

# Range Monitoring

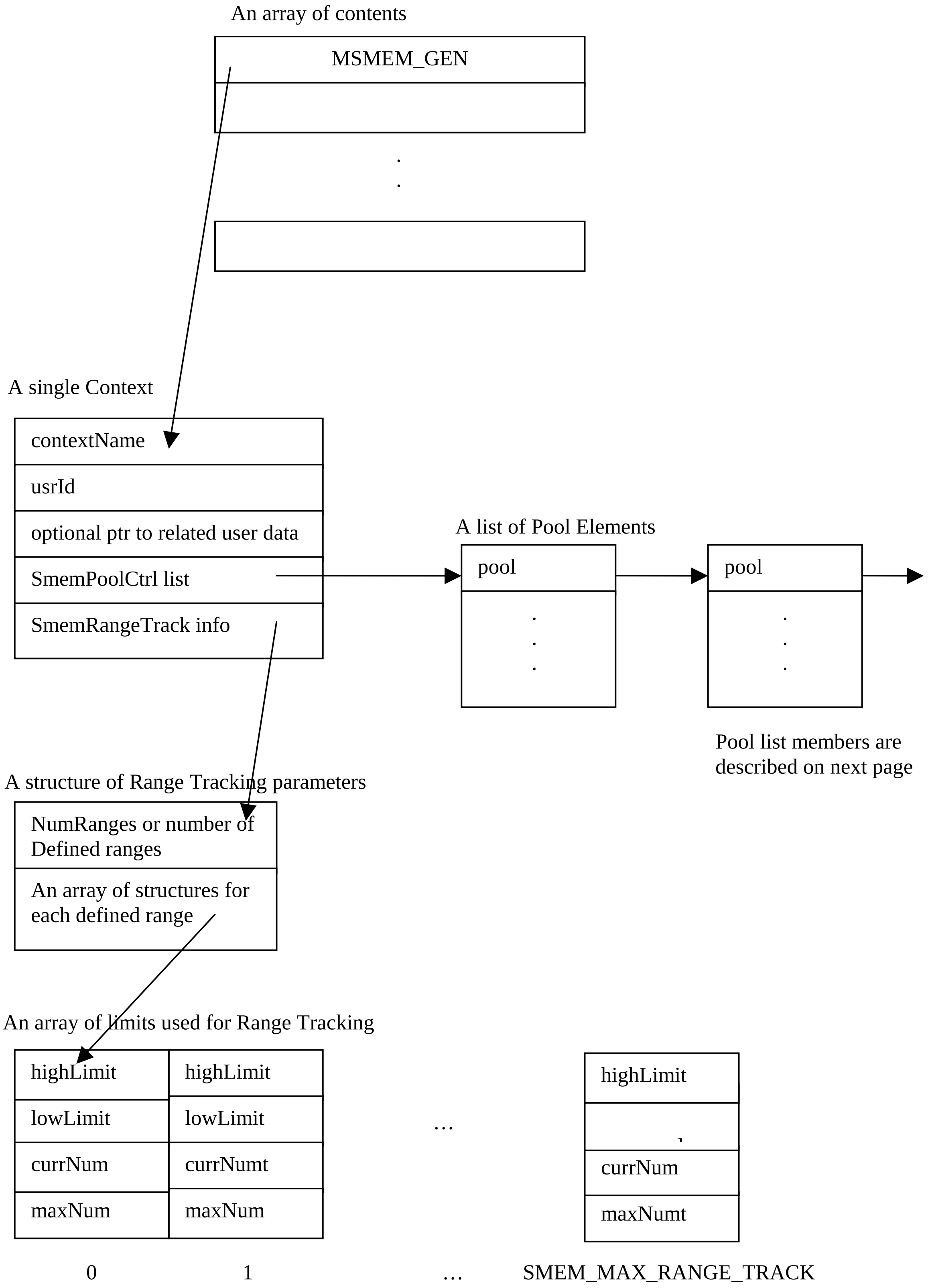
Up to **`SMEM_MAX_RANGE_TRACK`** (see define in **smem.h**) ranges of memory may be monitored to determine the amount of usage of each configured memory area. The ranges are defined by a high limit (**`HighLimit`**) and a low limit (**`LowLimit`**) entered in the configuration file for each specified range. These limits and the number of ranges to monitor are saved.

After configuration, during execution of the program, totals will be kept for the maximum number of memory buffers used in each range (**`maxNum`**) and the current number used (**`currNum`**). These totals are set to zero at startup.

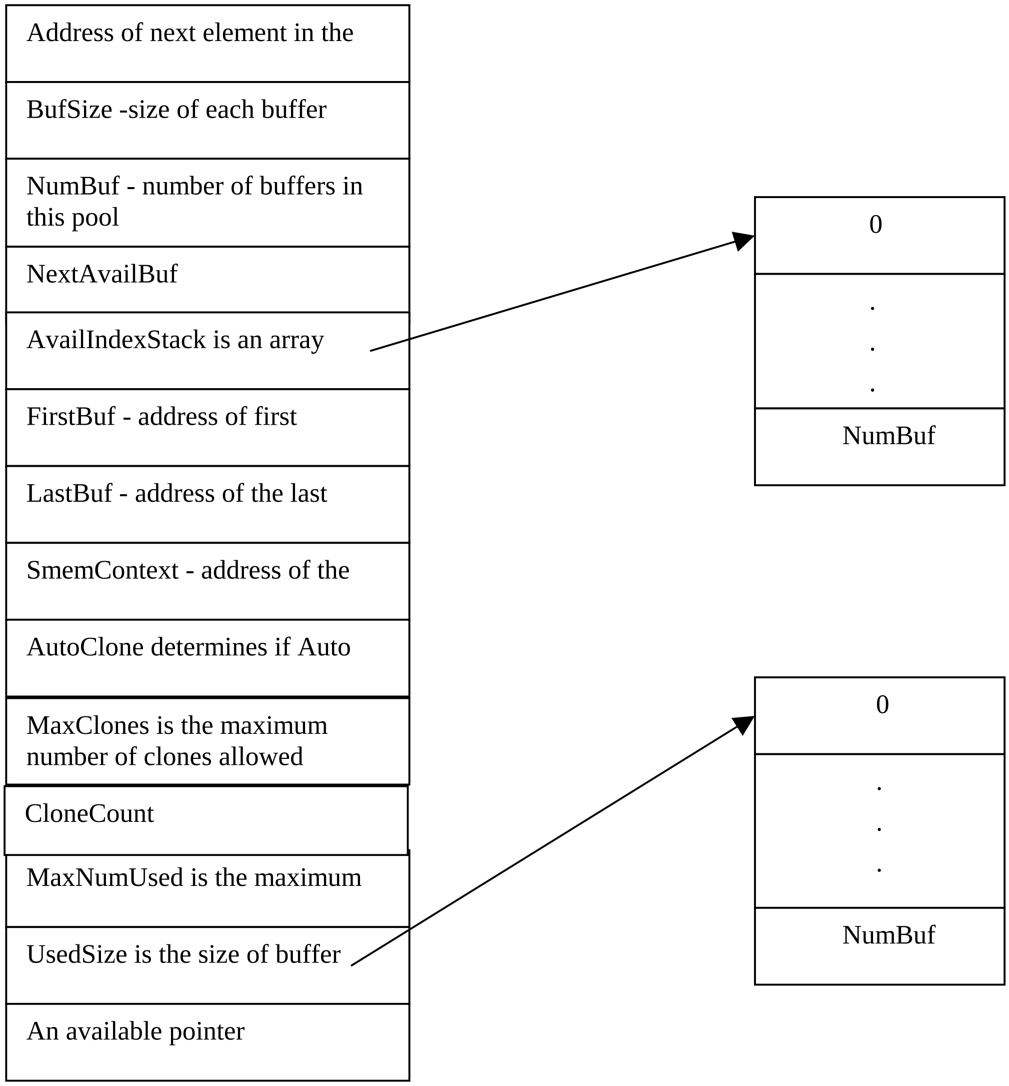
When a buffer is allocated in a SMEM context, its size is compared with each range configured. If it is inside a configured range then the count for that range is incremented. If the current count is larger than the maximum number used in this range so far, then the maximum is set to the current count.

When a buffer is freed in a SMEM context, its size is compared with each range configured. If it is inside a configured range, then the count for that range is decremented.

The following diagram of data structures is used:


An array of contents
MSMEM_GEN
A single Context
contextName
usrId
optional ptr to related user data
SmemPoolCtrl list
SmemRangeTrack info
A list of Pool Elements
pool
pool
Pool list members are described on next page
A structure of Range Tracking parameters
NumRanges or number of Defined ranges
An array of structures for each defined range
An array of limits used for Range Tracking
highLimit
lowLimit
currNum
maxNum
highLimit
lowLimit
currNumt
maxNum
…
highLimit
currNum
maxNumt
0
1
…
SMEM_MAX_RANGE_TRACK

An element in the list of available pools


Address of next element in the
BufSize -size of each buffer
NumBuf - number of buffers in this pool
NextAvailBuf
AvailIndexStack is an array
FirstBuf - address of first
LastBuf - address of the last
SmemContext - address of the
AutoClone determines if Auto
MaxClones is the maximum number of clones allowed
CloneCount
MaxNumUsed is the maximum
UsedSize is the size of buffer
An available pointer
0
.
.
.
NumBuf
0
.
.
.
NumBuf

AvailIndexStack array

## SMEM Data Type Definitions

The OPTIONAL “Range Tracking” feature uses the following structures.

```
typedef struct _smem_range_track
  {
  ST_UINT lowLimit;
  ST_UINT highLimit;
  ST_LONG currNum;
  ST_LONG maxNum;
  } SMEM_RANGE_TRACK;

typedef struct _smem_rt_ctrl
  {
  ST_INT numRanges;
  SMEM_RANGE_TRACK rt[SMEM_MAX_RANGE_TRACK];
  } SMEM_RT_CTRL;
```

The following structure is used to configure and control a specific SMEM Pool within a SMEM Context.

```
typedef struct _smem_pool_ctrl
  {
  struct _smem_pool_ctrl *next;
  ST_CHAR *poolName;          /* configuration parameter    */
  ST_UINT bufSize;            /* configuration parameter    */
  ST_UINT8 numBuf;            /* configuration parameter    */
  ST_UINT8 nextAvailBuf;      /* internal use   */
  ST_UINT8 *availIndexStack;  /* internal use   */
  ST_CHAR *firstBuf;          /* internal use   */
  ST_CHAR *lastBuf;           /* internal use   */
  struct _smem_context *smemContext;      /* context using this pool*/

  ST_BOOLEAN autoClone;       /* configuration parameter    */
  ST_INT maxClones;           /* configuration parameter    */
  ST_INT cloneCount;          /* internal use   */


#ifdef DEBUG_SISCO
  ST_UINT8 maxNumUsed;        /* internal use   */
  ST_UINT16 *usedSize;        /* internal use   */
#endif

  ST_VOID *usr;               /* SMEM user can use this ... */
  } SMEM_POOL_CTRL;
```

The following structure is the top-level structure containing all, important information about a SMEM Context.

```
typedef struct _smem_context
  {
  ST_CHAR *contextName; /* set automatically by smemcfgx to */
                        /* appropriate name in table below  */
  ST_INT   usrId;       /* configuration parameter    */
  ST_VOID *usr;         /* SMEM user can use this ... */

  SMEM_POOL_CTRL *smemPoolCtrlList; /* list of pools in this context*/

  /* Optional range tracking control structure  */
  SMEM_RT_CTRL *smemRangeTrack;
} SMEM_CONTEXT;
```

# SMEM Control Global Variables

SMEM is controlled by this global array of context control structures. Array elements 0 - 25 are used by the MMS-EASE *Lite* libraries. Array elements 26 -29 may be used by user code.

```
#define M_SMEM_MAX_CONTEXT    30
extern SMEM_CONTEXT m_smem_ctxt[M_SMEM_MAX_CONTEXT];
```

| CONTEXT NAME | **m_smem_ctxt ARRAY INDEX** |
|---|---|
| MSMEM_GEN | 0 |
| MSMEM_DEC_OS_INFO | 1 |
| MSMEM_ENC_OS_INFO | 2 |
| MSMEM_WR_DATA_DEC_BUF | 3 |
| MSMEM_ASN1_DATA_ENC | 4 |
| MSMEM_PDU_ENC | 5 |
| MSMEM_COM_EVENT | 6 |
| MSMEM_RXPDU | 7 |
| MSMEM_NETINFO | 8 |
| MSMEM_DYN_RT | 9 |
| MSMEM_AA_ENCODE | 10 |
| MSMEM_REQ_CTRL | 11 |
| MSMEM_IND_CTRL | 12 |
| MSMEM_MVLU_VA | 13 |
| MSMEM_MVLU_VA_CTRL | 14 |
| MSMEM_MVLU_VA_DATA | 15 |
| MSMEM_MVLU_GNL | 16 |
| MSMEM_MVLU_AA | 17 |
| MSMEM_ACSE_CONN | 18 |
| MSMEM_ACSE_DATA | 19 |
| MSMEM_COSP_CN | 20 |
| MSMEM_N_UNITDATA | 21 |
| MSMEM_SOCK_INFO | 22 |
| MSMEM_SPDU_TX | 23 |
| MSMEM_STARTUP | 24 |
| MSMEM_TPKT | 25 |

# SMEM Functions

## init_mem_chk

**Usage:** This function must be called before any other allocation functions to initialize the memory manager.

**Function Prototype:** `ST_VOID init_mem_chk (ST_VOID);`

**Parameters:** NONE

**Return Value:** `ST_VOID`

## smemcfgx

**Usage:** This function reads an XML file and uses the information to configure one or more SMEM contexts and optionally create one or more pools within each context. All data is stored in the **`m_smem_ctxt`** array of **`SMEM_CONTEXT`** structures and associated **`SMEM_POOL_CTRL`** structures. Contexts may be referenced by “Context Name” (see the table of Context Names above), or by “Context Index” (the index into the **`m_smem_ctxt`** array).

**Function Prototype:** `ST_RET smemcfgx (ST_CHAR *xml_filename);`

**Parameters:**

`xml_filename` Name of standard XML file containing the SMEM configuration information.

| | | | |
|---|---|---|---|
| **Return Value:** | `ST_RET` | `= SD_SUCCESS` | (configuration done) |
| | | `!= SD_SUCCESS` | (configuration failed) |

## u_smem_get_pool_params

**Usage:** This user function is called from **smemcfgx** if a pool is configured but either the "number of buffers" OR the "buffer size" is 0 (or not configured). This allows the user to determine at runtime the appropriate "buffer size" or "number of buffers" (possibly based on other configuration parameters. For example, the "number of buffers" may be based on the number of connections, or the "buffer size" may be based on the maximum MMS message size or the maximum TPDU size. Pointers to the current values of each parameter are passed to the function, so it is possible for the function to use the current values to compute new values and then to write the new values at the pointer location.

**Function Prototype:**

```
ST_VOID  u_smem_get_pool_params (SMEM_CONTEXT *smemContext,
                                 ST_UINT8 *numBuf,
                                 ST_UINT *bufSize,
                                 ST_BOOLEAN *autoClone,
                                 ST_INT *maxClones,
                                 ST_CHAR **poolName);
```

**Parameters:**

| | |
|---|---|
| `smemContext` | Pointer to a SMEM context containing this pool. This will always be a pointer to an element of the **m_smem_ctxt** global array of contexts. |
| `numBuf` | Pointer to the "number of  buffers" in the pool. |
| `bufSize` | Pointer to the "buffer size". |
| `autoClone` | Pointer to flag to indicate if this pool should be automatically cloned when it runs out of buffers. |
| `maxClones` | Pointer to maximum number of clones to create if automatic cloning is enabled. |
| `poolName` | Pointer to pointer to optional pool name string. |

**Return Value:** `ST_VOID`

## u_smem_need_buffers

**Usage:** This user function is called when a SMEM context has no more available buffers of a particular buffer size. The user must create an appropriate pool by calling **smem_add_pool** and return a pointer to the new pool control structure.

**Function Prototype:**

```
SMEM_POOL_CTRL *u_smem_need_buffers (SMEM_CONTEXT *smemContext,
                                     ST_UINT8 numBuf,
                                     ST_UINT bufSize);
```

**Parameters:**

| | |
|---|---|
| smemContext | Pointer to a SMEM context containing this pool. This will always be a pointer to an element of the **m_smem_ctxt** global array of contexts. |
| numBuf | Number of buffers in an existing pool for this buffer size or 0 if no pools exist for this buffer size. |
| bufSize | Size of buffers needed. This buffer size (or a larger value) should be passed to **smem_add_pool**. |

***WARNING***: DO NOT pass numBuf = 0 or bufSize = 0 to **smem_add_pool** (these are not legal values).

**Return Value:** (SMEM_POOL_CTRL *) Pointer to new pool created or **NULL** if pool could not be created.

## smem_add_pool

**Usage:** This function adds a new pool to a SMEM context.

**Function Prototype:**

```
SMEM_POOL_CTRL *smem_add_pool (SMEM_CONTEXT *smemContext,
                               ST_UINT8 numBuf,
                               ST_UINT bufSize,
                               ST_BOOLEAN autoClone,
                               ST_INT maxClones,
                               ST_CHAR *poolName);
```

**Parameters:**

| | |
|---|---|
| `smemContext` | Pointer to a SMEM context containing this pool. This MUST be a pointer to an element of the **`m_smem_ctxt`** global array of contexts. |
| `numBuf` | Number of buffers in the pool. |
| `bufSize` | Buffer size. |
| `autoClone` | Flag to indicate if this pool should be automatically cloned when it runs out of buffers. |
| `maxClones` | Maximum number of clones to create if automatic cloning is enabled. |
| `poolName` | Optional pool name string. |

**Return Value:** `(SMEM_POOL_CTRL *)` Pointer to new pool created or **`NULL`** if pool could not be created.

## smem_log_state

**Usage:** This function writes the current state of the SMEM context to the log file, including all pools and buffers currently in use. This information may help determine if the pools configured in this context are appropriate.

**Function Prototype:** `ST_VOID smem_log_state (SMEM_CONTEXT *smemContext);`

**Parameters:**

| | |
|---|---|
| `smemContext` | Pointer to a SMEM context containing this pool. This MUST be a pointer to an element of the **`m_smem_ctxt`** global array of contexts. |

**Return Value:** `ST_VOID`

## m_add_pool

**Usage:** This function adds a new pool to a SMEM context. This function is exactly the same as **`smem_add_pool`** except it adds overhead to each buffer for additional tracking information such as the file and line number where the buffer was allocated.

**Function Prototype:**

```
SMEM_POOL_CTRL *m_add_pool (SMEM_CONTEXT *smemContext,
                            ST_UINT8 numBuf,
                            ST_UINT bufSize,
                            ST_BOOLEAN autoClone,
                            ST_INT maxClones,
                            ST_CHAR *poolName);
```

**Parameters:**

| | |
|---|---|
| `smemContext` | Pointer to a SMEM context containing this pool. This MUST be a pointer to an element of the **`m_smem_ctxt`** global array of contexts. |
| `numBuf` | Number of  buffers in the pool. |
| `bufSize` | Buffer size. Overhead will be added to this size. |
| `autoClone` | Flag to indicate if this pool should be automatically cloned when it runs out of buffers. |
| `maxClones` | Maximum number of clones to create if automatic cloning is enabled. |
| `poolName` | Optional pool name string. |

**Return Value:** `(SMEM_POOL_CTRL *)` Pointer to new pool created or **`NULL`** if pool could not be created.

## x_m_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 are 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 *x_m_malloc (SMEM_CONTEXT *ctx,
                     ST_UINT size,
                     ST_CHAR *file,
                     ST_INT line);
```

**Parameters**:

| | |
|---|---|
| `ctx` | Context from which to allocate buffer. |
| `size` | This indicates the size in bytes of the buffer to be allocated. |
| `file` | Name of source file where this function is called. |
| `line` | Line number in source file where this function is called. |

| | | | |
|---|---|---|---|
| **Return Value:** | `ST_VOID *` | `<> null` | This is a pointer to the allocated buffer. |
| | | `= null` | The memory allocation has failed. |

## x_m_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 *x_m_calloc (SMEM_CONTEXT *ctx,
                     ST_UINT num,
                     ST_UINT size,
                     ST_CHAR *file,
                     ST_INT line);
```

**Parameters:**

| | |
|---|---|
| ctx | Context from which to allocate buffer. |
| 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. |
| file | Name of source file where this function is called. |
| line | Line number in source file where this function is called. |

| **Return Value:** | | | |
|---|---|---|---|
| | ST_VOID * | <> null | This is a pointer to the allocated buffer. |
| | | = null | The memory allocation failed. |

## x_m_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 *x_m_realloc (SMEM_CONTEXT *ctx,
                      ST_VOID *old,
                      ST_UINT size,
                      ST_CHAR *file,
                      ST_INT line);
```

**Parameters:**

| | |
|---|---|
| `ctx` | Context from which to reallocate buffer. |
| `old` | This pointer indicates the old buffer. |
| `size` | This indicates the new size of the buffer. |
| `file` | Name of source file where this function is called. |
| `line` | Line number in source file where this function is called. |

| | | | |
|---|---|---|---|
| **Return Value:** | `ST_VOID *` | `<> null` | This is a pointer to the resized buffer. |
| | | `= null` | The memory reallocation failed. |

## x_m_free

**Usage:** This function deallocates a memory buffer allocated with **`x_m_calloc`**, **`x_m_malloc`**, or **`x_m_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 x_m_free (SMEM_CONTEXT *ctx,
                  ST_VOID *ptr,
                  ST_CHAR *file,
                  ST_INT line);
```

**Parameters**:

| | |
|---|---|
| `ctx` | Context from which to deallocate buffer. |
| `ptr` | This is a pointer to the memory buffer that is to be deallocated. |
| `file` | Name of source file where this function is called. |
| `line` | Line number in source file where this function is called. |

**Return Value:** `ST_VOID`

# Changes Required to Use SMEM

The libraries and sample applications provided with MMS-EASE *Lite* use the SISCO **MEM_CHK** Memory Manager by default. The source code and build process must be modified as follows to use the SISCO SMEM Memory Manager instead.

## *Source Code Changes*

1. The sample code in **uca_srvr.c** demonstrates these changes (search for **#ifdef SMEM_ENABLE**).
2. At startup, **init_mem_chk** must be called BEFORE any memory allocation function (including **smemcfgx**) is called. If it is not called, the application will exit due to an assertion.
3. Also at startup, **smemcfgx** should be called (after **init_mem_chk**) to configure memory contexts and pools. If this function is not called, there will be no memory pools configured at startup. In this case, when a memory allocation function is called, the user function **u_smem_need_buffers** (see below) will be called to allow the user to create a pool as needed.
4. The user functions **u_smem_get_pool_params** and **u_smem_need_buffers** must be written (see examples in **uca_srvr.c**).
5. Before exiting the application, **smem_log_state** should be called for each SMEM context, to write to the log file detailed information about the current and the maximum memory used in each context.

## *Build Process Changes*

1. Build the Foundry executable **before** making any changes. This program is not written to use SMEM.
2. If using Microsoft Windows and Microsoft Visual C++, change the “Project Dependencies” so that the application you wish to build does NOT depend on the Foundry. This will prevent it from being rebuilt when you select “Rebuild All” to rebuild the application.
3. Define **SMEM_ENABLE** in the makefile (or DSP file) for each library and application to be built.
4. Change the linker command in the makefile (or DSP file) for the application, to link to **smem.lib** (**smemd.lib** for DEBUG) instead of **mem.lib** (**memd.lib** for DEBUG).
5. Rebuild the application and all associated libraries.

# Recommended SMEM Configuration Procedure

The following procedure is recommended for obtaining the optimum SMEM configuration.

1. Make sure that the application is compiled with **DEBUG_SISCO** defined and that, just before exiting, it calls **smem_log_state** for each context (see **uca_srvr.c**).
2. Create an empty **smemcfg.xml** file so that no SMEM contexts or pools are configured.
3. Run the application, preferably under extreme conditions (e.g. maximum number of connections, transferring large amounts of data, etc.). With no SMEM configuration, the application will call the user function **u_smem_need_buffers** when it needs a new SMEM pool. This will usually NOT create optimum pools.

4. Exit the application and examine the log file. It should contain details about each SMEM pool created, including the maximum number of buffers used in each pool, and the number of bytes actually used in each buffer. For example, if there is a pool containing unused buffers that are 8000 bytes, and an allocation function requests 200 bytes, SMEM will use one of the 8000 byte buffers, wasting 7800 bytes.
5. Edit the **smemcfg.xml** file to configure pools that contain ONLY the number of buffers and the buffer sizes that are actually needed.
6. Run the application again. If the configuration is optimum, all buffers in all pools will be used at some time, and the user function **`u_smem_need_buffers`** will NEVER be called. Use a debugger or a printf to determine if **`u_smem_need_buffers`** is ever called.
7. If necessary, edit the configuration and run the application again until the optimum pools are created.
8. Run the application under all reasonable conditions to be sure that the configured pools are adequate.

## Appendix E

# GLBSEM Subsystem for Multi-Threaded Support

This section addresses the issues related to writing a thread-safe MMS-EASE application.

To support thread-safe 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.

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.

Sample code that shows how to use the MMS-EASE multi-threaded API is available from Technical Support upon request.

***IMPORTANT:*** *These functions and macros are only available on operating systems that support muli-threading. They may need to be "ported" to your system.*

## *SISCO's Global Mutex (Mutual Exclusion) Semaphore Macros*

### S_LOCK_COMMON_RESOURCES

**Usage:** Gives the current thread exclusive access to MMS global objects.

**Function Prototype:** `S_LOCK_COMMON_RESOURCES ();`

**Parameters**: NONE

**Return Value:** NONE

### S_UNLOCK_COMMON_RESOURCES

**Usage:** Releases exclusive access to MMS global objects.

**Function Prototype:** `S_UNLOCK_COMMON_RESOURCES ();`

**Parameters**: NONE

**Return Value:** NONE

## *Mutex Semaphore Functions*

### gs_mutex_create

**Usage:** This function creates a mutex semaphore.

**Function Prototype:** `ST_RET gs_mutex_create (ST_MUTEX_SEM *ms);`

**Parameters:**

| | |
|---|---|
| `ms` | This is a pointer to **`ST_MUTEX_SEM`** where information about the created mutex semaphore is stored. |

**Return Value:** `SD_RET` `SD_SUCCESS` or `SD_FAILURE`

### gs_mutex_get

**Usage:** This function obtains ownership of a mutex semaphore.

**Function Prototype:** `ST_VOID gs_mutex_get (ST_MUTEX_SEM *ms);`

**Parameters:**

| | |
|---|---|
| `ms` | This is a pointer to the mutex semaphore. |

**Return Value:** `ST_VOID`

### gs_mutex_free

**Usage:** This function releases ownership of a mutex semaphore.

**Function Prototype:** `ST_VOID gs_mutex_free (ST_MUTEX_SEM *ms);`

**Parameters:**

| | |
|---|---|
| `ms` | This is a pointer to the mutex semaphore. |

**Return Value:** `ST_VOID`

## gs_mutex_destroy

**Usage:** This function destroys the mutex semaphore.

**Function Prototype:** `ST_VOID gs_mutex_destroy (ST_MUTEX_SEM *ms);`

**Parameters:**

| | |
|---|---|
| `ms` | This is a pointer to the mutex semaphore. |

| | | |
|---|---|---|
| **Return Value:** | `SD_RET` | `SD_SUCCESS` or `SD_FAILURE` |

# *Event Semaphore Functions*

## gs_get_event_sem

**Usage:** This function creates and initializes a manual-reset or auto-reset event semaphore.

**Function Prototype:** `ST_EVENT_SEM gs_get_event_sem (ST_BOOLEAN manualReset);`

**Parameters:**

| | |
|---|---|
| `manualReset` | This is a boolean flag that is set to **`SD_TRUE`** or **`SD_FALSE`**. |

| | | |
|---|---|---|
| **Return Value:** | `ST_EVENT_SEM` | This is a handle to an event semaphore. |

## 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`

**Notes:**

**Manual-reset event semaphore:**
When you use **`gs_signal_event_sem`**, all waiting threads are released, and the event remains in signaled state until you explicitly reset it using **`gs_reset_event_sem`**.

**Auto-reset event semaphore:**
When you use **`gs_signal_event_sem`**, only the first waiting thread is released, and the event is reset to non-signaled state before the function returns. However, if no thread is waiting, the state remains signaled unless reset explicitly using **`gs_reset_event_sem`**.

## gs_pulse_event_sem

**Usage:** This function is used to pulse an event semaphore.

**Function Prototype:** `ST_VOID gs_pulse_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`

**Notes:**

**Manual-reset event semaphore:**
When you use **`gs_pulse_event_sem`**, all waiting threads are released, and the event's state is reset to non-signaled before the function returns.

**Auto-reset event semaphore:**
When you use **`gs_pulse_event_sem`**, only the first waiting thread is released, and the event is reset to non-signaled state before the function returns, even if there are no waiting threads.

## 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. If the timeout is greater than 0, the function waits for the event semaphore for the duration of the timeout period

| | | | |
|---|---|---|---|
| **Return Value:** | SD_RET | SD_SUCCESS | The semaphore is signaled. |
| | | SD_TIMEOUT | The timeout period elapsed and the semaphore is non-signaled. |
| | | SD_FAILURE | Any other error condition. |

## gs_wait_mult_event_sem

**Usage:** This function is implemented on Windows systems only. It is used to check the state of multiple event semaphores. If the state of a semaphore is signaled, the function returns immediately. Otherwise, it blocks the caller until either a semaphore is signaled or a timeout occurs.

**Function Prototype:**

```
ST_RET gs_wait_mult_event_sem (ST_INT numEvents,
                               ST_EVENT_SEM *esTable,
                               ST_BOOLEAN *activity,
                               ST_LONG timeout)
```

**Parameters:**

| | |
|---|---|
| `numEvents` | This is the number of event semaphores to wait for. |
| `esTable` | This is a pointer to a table of event semaphore objects. |
| `activity` | This is a pointer to a table where this function will mark a proper index entry with **`SD_TRUE`** for the event semaphore that have been signaled. |
| `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. If the timeout is greater than 0, the function waits for the event semaphore for the duration of the timeout period |

| | | | |
|---|---|---|---|
| **Return Value:** | `SD_RET` | `SD_SUCCESS` | The semaphore is signaled. |
| | | `SD_TIMEOUT` | The timeout period elapsed and the semaphore is non-signaled. |
| | | `SD_FAILURE` | Any other error condition. |

## gs_reset_event_sem

**Usage:** This function is used to reset a manual-reset event semaphore. Call this function only if the function **`gs_wait_event_sem`** returns **`SD_SUCCESS`**. If **`gs_wait_mult_event_sem`** is used, this function should be called for every manual-reset semaphore with the **`activity`** table entry set to **`SD_TRUE`**.

**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`

## 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_get_event_sem`**.

**Return Value:** `ST_VOID`

## *Thread Functions*

### gs_start_thread

**Usage:** This function starts a new thread.

**Function Prototype:**

```
ST_RET gs_start_thread (ST_THREAD_RET (ST_THREAD_CALL_CONV *threadFunc)
                                      (ST_THREAD_ARG),
                                      ST_THREAD_ARG threadArg,
                                      ST_THREAD_HANDLE threadHandleOut,
                                      ST_THREAD_ID *threadIdOut);
```

**Parameters:**

| | |
|---|---|
| `threadFunc` | This is a pointer to thread function to run. |
| `threadArg` | This is a thread function argument list. |
| `threadHandleOut` | This is a pointer where to return the thread handle. |
| `threadIdOut` | This is a pointer where to return the thread ID. |

**Return Value:** `SD_RET` `SD_SUCCESS` or `SD_FAILURE`

### gs_wait_thread

**Usage:** This function waits until the thread with **`threadHandle`** terminates or timeout occurrs.

On UNIX systems, there is no option for timed wait. This function will wait until the thread is terminated.

**Function Prototype:**

```
ST_RET gs_wait_thread (ST_THREAD_HANDLE threadHandle,
                       ST_THREAD_ID threadId,
                       ST_LONG timeout);
```

**Parameters:**

| | |
|---|---|
| `threadHandle` | This is the thread handle returned from **`gs_start_thread`**. |
| `threadId` | This is the thread ID returned from **`gs_start_thread`**. |
| `timeout` | This is the maximum time in milliseconds to wait for the thread to terminate. |

**Return Value:** `SD_RET` `SD_SUCCESS` or `SD_FAILURE`

## gs_close_thread

**Usage:** This function releases resources for the terminated thread.

**Function Prototype:** `ST_RET gs_close_thread (ST_THREAD_HANDLE threadHandle);`

**Parameters:**

`threadHandle` This is the thread handle returned from **`gs_start_thread`**.

**Return Value:** `SD_RET` `SD_SUCCESS` or `SD_FAILURE`

## gs_start_thread

**Usage:** This function starts a new thread.

**Function Prototype:**

```
ST_RET gs_start_thread (ST_THREAD_RET (ST_THREAD_CALL_CONV *threadFunc)
                                       (ST_THREAD_ARG),
                                       ST_THREAD_ARG threadArg,
                                       ST_THREAD_HANDLE threadHandleOut,
                                       ST_THREAD_ID *threadIdOut);
```

**Parameters:**

`threadFunc` This is a pointer to thread function to run.

`threadArg` This is a thread function argument list.

`threadHandleOut` This is a pointer where to return the thread handle.

`threadIdOut` This is a pointer where to return the thread ID.

**Return Value:** `SD_RET` `SD_SUCCESS` or `SD_FAILURE`

## Appendix F

# Support Functions

This section contains all of what are considered miscellaneous functions of MMS-EASE *Lite*.

# UCT Time Support Functions

The following structure is used by UCT Time support functions shown below.

```
/* Binary Time Of Day                                       */
#define MMS_BTOD4 4
#define MMS_BTOD6 6

typedef struct btod_data
  {
  ST_INT form;    /* MMS_BTOD6, MMS_BTOD4                   */
  ST_INT32 ms;    /* Number of milliseconds since midnight  */
  ST_INT32 day;   /* Number of days since Jan 1, 1984       */
  } MMS_BTOD;
```

The following structure is used to store the UCT Time MMS type.

```
typedef struct mms_utc_time_tag
  {
  ST_UINT32 secs;   /* Number of seconds since GMT midnight January 1, 1970  */
  ST_UINT32 usec;   /* Number of microseconds of a second                    */
  ST_UINT32 qflags; /* Quality flags, 8 least-significant bits only          */
  } MMS_UTC_TIME;
```

### asn1_convert_btod_to_utc

**Usage:** This function converts **`MMS_BTOD`** (time relative to 1/1/1984) to the **`MMS_UTC_TIME`** (time relative to 1/1/1970). The qflags field in the **`MMS_UTC_TIME`** needs to be set by the calling function. Only the **`MMS_BTOD6`** form of the **`MMS_BTOD`** struct can be converted to the **`MMS_UTC_TIME`**.

**Function Prototype:**
```
ST_RET asn1_convert_btod_to_utc (MMS_BTOD *btod,
                                 MMS_UTC_TIME *utc);
```

**Parameters**:

| | |
|---|---|
| `btod` | This is a pointer to **`MMS_BTOD`** struct that should be converted to the **`MMS_UTC_TIME`**. |
| `utc` | This is a pointer to **`MMS_UTC_TIME`** structure where the result of the conversion will be placed. |

**Return Value:** `ST_RET` `SD_SUCCESS` or `SD_FAILURE`

## asn1_convert_utc_to_btod

**Usage:** This function converts **MMS_UTC_TIME** (time relative to 1/1/1970) to the **MMS_BTOD** (time relative to 1/1/1984). The form field in the **MMS_BTOD** is set to **MMS_BTOD6** by this function.

**Function Prototype:**

```
ST_RET asn1_convert_utc_to_btod (MMS_UTC_TIME *utc,
                                 MMS_BTOD *btod);
```

**Parameters**:

| | |
|---|---|
| utc | This is a pointer to **MMS_UTC_TIME** struct that should be converted to the **MMS_BTOD**. |
| btod | This is a pointer to **MMS_BTOD** struct where the result of the conversion will be placed. |

**Return Value:** ST_RET SD_SUCCESS or SD_FAILURE

## Appendix G

# Subnetwork API

The Subnetwork Layer’s purpose is to provide a consistent interface to be used by the CLNP layer. Because the LLC layer is included in the CLNP Layer, the CLNP layer could interface directly to the MAC API (ADLC, Ethernet, etc.). However, this would require the CLNP Layer to be modified to interface to each MAC API (which vary significantly for different MAC layers and different operating systems). To avoid this, the Subnetwork layer is inserted. It provides a single Subnetwork API that is used by the CLNP layer. It performs the operations necessary to translate the Subnetwork API commands into the appropriate MAC API commands. Thus, porting to a new MAC layer requires only rewriting the Subnetwork API functions described below.

## Subnetwork Data Structure

This structure below is used to write packets to the Subnetwork and to read packets from the Subnetwork.

```
typedef struct
  {
  ST_UCHAR   loc_mac [CLNP_MAX_LEN_MAC];
  ST_UCHAR   rem_mac [CLNP_MAX_LEN_MAC];
  ST_UINT16  lpdu_len;
  ST_UCHAR   *lpdu;
  ST_BOOLEAN free_lpdu;
  }SN_UNITDATA;
```

**<u>Fields:</u>**

`loc_mac` This is the buffer for the local MAC address. Its length is determined by `CLNP_MAX_LEN_MAC`.

`rem_mac` This is the buffer for the remote MAC address. Its length is determined by `CLNP_MAX_LEN_MAC`.

`lpdu_len` This is the length of the field **`lpdu`**.

`lpdu` This is a pointer to the **`lpdu`** buffer to send.

`free_lpdu` This flag indicates whether the **`lpdu`** buffer should be freed. If `SD_TRUE`, the **`lpdu`** buffer is deallocated by the **`clnp_read`** function.

## Compile Time Options

The following is and optional compile time option:

`-DDEBUG_SISCO` Enable logging using “slog”

## *Subnetwork Functions*

### clnp_snet_init

**Usage:** This function will initialize the subnetwork layer.

**Function Prototype:** `ST_RET clnp_snet_init (CLNP_PARAM *clnp_param);`

**Parameters:**

| | |
|---|---|
| `clnp_param` | This is a pointer to a structure containing CLNP configuration parameters. |

| | | | |
|---|---|---|---|
| **Return Value:** | `ST_RET` | `SD_SUCCESS` | No Error |
| | | `! = SD_SUCCESS` | Error |

### clnp_snet_term

**Usage:** This function will terminate the subnetwork layer.

**Function Prototype:** `ST_RET clnp_snet_term (ST_VOID);`

**Parameters:** NONE

| | | | |
|---|---|---|---|
| **Return Value:** | `ST_RET` | `SD_SUCCESS` | No Error |
| | | `!= SD_SUCCESS` | Error |

### clnp_snet_read

**Usage:** This function will receive a LPDU from a subnetwork.

**Function Prototype:** `ST_RET clnp_snet_read (SN_UNITDATA *sn_req);`

**Parameters:**

| | |
|---|---|
| `sn_req` | This is a pointer of structure type `SN_UNITDATA` to the Subnetwork Unit Data request to be received. |

| | | | |
|---|---|---|---|
| **Return Value:** | `ST_RET` | `= SD_SUCCESS` | No Error |
| | | `!= SD_SUCCESS` | Error |

## clnp_snet_write

**Usage:** This function will send a LPDU to a subnetwork.

**Function Prototype:** `ST_RET clnp_snet_write (SN_UNITDATA *sn_req);`

**Parameters:**

| | |
|---|---|
| `sn_req` | This is a pointer of structure type **`SN_UNITDATA`** to a Subnetwork Unit Data request to be sent. |

| | | | |
|---|---|---|---|
| **Return Value:** | `ST_RET` | `= SD_SUCCESS` | No Error |
| | | `!= SD_SUCCESS` | Error |

## clnp_snet_free

**Usage:** This function frees up subnetwork resources associated with a received SN-UNITDATA PDU.

**Function Prototype:** `ST_VOID clnp_snet_free (SN_UNITDATA *sn_req);`

**Parameters:**

| | |
|---|---|
| `sn_req` | This is a pointer to a structure containing information about the SN-UNITDATA PDU received. |

**Return Value:** `NONE`

## clnp_snet_get_local_mac

**Usage:** This function will copy to the buffer **`mac_buf`** the local MAC address for a given subnetwork.

**Function Prototype:** `ST_RET clnp_snet_get_local_mac (ST_UCHAR *mac_buf);`

**Parameters:**

| | |
|---|---|
| `mac_buf` | This is a pointer to the buffer for MAC address. The buffer is at least `CLNP_MAX_LEN_MAC` bytes long. |

| | | | |
|---|---|---|---|
| **Return Value:** | `ST_RET` | `= SD_SUCCESS` | No Error |
| | | `!= SD_SUCCESS` | Error |

## clnp_snet_set_multicast_filter

**Usage:** This function enables the reception of multicast packets by the Ethernet driver. Multicast packets include GOOSE messages and ES-IS protocol packets required by the OSI stack. The driver will accept incoming packets in which the destination MAC address matches one of these multicast MAC addresses. If the Ethernet driver is already set to promiscuous mode, this function does not need to be called.

**Function Prototype:**

```
ST_RET clnp_snet_set_multicast_filter (ST_UCHAR *mac_list,
                                       ST_INT num_macs);
```

**Parameters:**

`mac_list` This is a pointer to a set of multicast MAC addresses (6 bytes each) on which to accept incoming packets.

`num_macs` This is the number of MAC addresses contained in **`mac_list`**.

**Return Value:**

| | | | |
|---|---|---|---|
| | ST_RET | = SD_SUCCESS | No Error |
| | | != SD_SUCCESS | Error |

**Comments:** This function may not be called until AFTER **`clnp_snet_init`** (or **`mvl_start_acse`**) is called.

## clnp_snet_add_multicast_mac

**Usage:** This function is provided for backward compatibility only. It may overwrite the existing list of multicast addresses. It is recommended that you use **`clnp_snet_set_multicast_filter`** instead.

This function will add the multicast MAC address in **`mac_buf`** to the set of multicast addresses on which to accept incoming packets.

**Function Prototype:**

```
ST_RET clnp_snet_add_multicast_mac (ST_UCHAR *mac_buf);
```

**Parameters:**

`mac_buf` This is a pointer to the multicast MAC address on which to accept incoming packets. The buffer is at least **`CLNP_MAX_LEN_MAC`** bytes long.

**Return Value:**

| | | | |
|---|---|---|---|
| | ST_RET | = SD_SUCCESS | No Error |
| | | != SD_SUCCESS | Error |

## clnp_snet_rx_all_multicast_start

**Usage:** This function enables the reception of "ALL multicast" packets by the Ethernet driver so that ALL incoming multicast packets are accepted. Multicast packets include GOOSE messages and ES-IS protocol packets required by the OSI stack. The driver remains in this mode until **`clnp_snet_rx_all_multicast_stop`** is called.

**Function Prototype:** `ST_RET clnp_snet_rx_all_multicast_start (ST_VOID);`

**Parameters:** NONE

| | | | |
|---|---|---|---|
| **Return Value:** | `ST_RET` | `= SD_SUCCESS` | Completed successfully. |
| | | `!= SD_SUCCESS` | Error code. |

**Comments:** This function may not be called until AFTER **`clnp_snet_init`** (or **`mvl_start_acse`**) is called.

## clnp_snet_rx_all_multicast_stop

**Usage:** This function disables the reception of "ALL multicast" packets by the Ethernet driver. It will continue accepting multicast packets that were "subscribed" for using **`clnp_snet_set_multicast_filter`**.

**Function Prototype:** `ST_RET clnp_snet_rx_all_multicast_stop (ST_VOID);`

**Parameters:** NONE

| | | | |
|---|---|---|---|
| **Return Value:** | `ST_RET` | `= SD_SUCCESS` | Completed successfully. |
| | | `!= SD_SUCCESS` | Error code. |

**Comments:** This function may not be called until AFTER **`clnp_snet_init`** (or **`mvl_start_acse`**) is called.

## clnp_snet_get_max_udata_len

**Usage:** This function will return the maximum length of user data for a given subnetwork.

**Function Prototype:** `ST_UINT16 clnp_snet_get_max_udata_len (ST_VOID);`

| | | |
|---|---|---|
| **Parameters:** | NONE | |
| **Return Value:** | `ST_UNIT16` | This returns the maximum length of the user data. |

## clnp_snet_get_type

**Usage:** This function will return the subnetwork type.

**Function Prototype:** `ST_INT clnp_snet_get_type (ST_VOID);`

| | | |
|---|---|---|
| **Parameters:** | NONE | |
| **Return Value:** | `ST_INT` | `SUBNET_ADLC` (SISCO ADLC Subnet)<br>`SUBNET_ETHE` (SISCO Ethernet Subnet) |

**Note:** If a new Subnetwork type is created, a new define should be added to **clnp_sne.h** to identify it. This is where the Subnetwork defines are stored.

Porting of the Subnetwork code to use a new MAC API (for a new operating system or new MAC layer), usually requires rewriting all of the Subnetwork API functions described above. SISCO provides examples of Subnetwork API functions to interface to SISCO's ADLC MAC (in **clnp_sne.c**) or to interface to a typical Ethernet NDIS MAC driver (in **clnp_eth.c**). This code must be modified if a different MAC API must be used.

## clnp_snet_check_mac

**Usage:** This function examines the MAC address referenced by **`mac_buf`** and returns a value indicating if it is the local MAC address, the ALL-ES Multicast MAC address, etc.

**Function Prototype:** `ST_INT clnp_snet_check_mac (ST_UINT8 *mac_addr);`

**Parameters:**

| | |
|---|---|
| `mac_buf` | Pointer to sequence of bytes representing MAC address. |

| | | |
|---|---|---|
| **Return Value:** | `ST_INT` | `CLNP_MAC_LOCAL` (address of this computer) |
| | | `CLNP_MAC_ALL_ES` (All-ES Multicast address) |
| | | `CLNP_MAC_GOOSE` (Other Multicast address. Probably GOOSE.) |
| | | `CLNP_MAC_INVALID` (Unrecognized address) |

## Appendix H

# MMS-EASE Type Description Language (TDL)

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.

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 [1..2]. A **`ST_INT16`** 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 **Int64** 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 +$2^{64}$ -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.

**Utctime** This type is encoded as UtcTime with seconds relative to GMT midnight January 1, 1970. The SISCO macro for the C language representation of Utctime is a structure (**`MMS_UTC_TIME`**) containing 3 consecutive **`ST_UINT32`**. The value contained in the first **`ST_UINT32`** represents the number of seconds since January 1, 1970. The seconds **`ST_UINT32`** represents number of microseconds of a second. And the last **`ST_UINT32`** contains quality flags, only least significant byte is used.

**VstringXXX** This type is encoded as a MMS visible string of a variable length not to exceed XXX 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 (OstringXXX) instead. The SISCO macro for the C language representation of **VstringXXX** is **`ST_CHAR [XXX+1]`**, where XXX 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 XXX, is the maximum length that the variable can be. MMS-EASE only sends or receives data up to a null or XXX bytes for variables of this type. For example, “Vstring24” specifies a VisibleString with no more than 24 characters.

**FstringXXX** This type is encoded as a MMS visible string of a fixed length of XXX 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 (OstringXXX) instead. The C language representation of **FstringXXX** is **`ST_CHAR [XXX+1]`**, where XXX 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 XXX 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 XXX bytes. For example, Fstring16 specifies a fixed length VisibleString consisting of exactly 16 characters.

**OstringXXX** This type is encoded as a MMS OctetString of a fixed length of XXX 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 **OstringXXX** is **`ST_UCHAR [XXX]`**, where XXX 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 XXX is the actual length that will be sent on the wire. For example, Ostring256 specifies a data stream of exactly 256 bytes.

**OVstringXXX** This type is encoded as a MMS OctetString of a variable length not to exceed XXX 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 XXX 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 **OVstringXXX** is:

```
struct  ovstring {
   ST_INT16  len;
   ST_UCHAR  data[XXX];
   };
```

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 XXX. Note that there is no extra byte for the null because a null can be a valid member of an OctetString.

**BstringXXX** This type is encoded as a MMS BitString of a fixed length of XXX bits. The SISCO macro for the C language representation of **BstringXXX** 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 XXX, 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.

**BVstringXXX** This type is encoded as a MMS BitString of a variable length of not to exceed XXX 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 XXX, 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 **BVstringXXX** is shown below:

```
struct  bvstring {
   ST_INT16  len;
   ST_UCHAR  data[YYY];
   };
```

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 XXX. YYY is the number of bytes in the array equal to (XXX+7)/8.

# TDL 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 the 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).

# TDL 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]}
   ```

**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:

The type definition for test1, test2, and test3 has already been created.

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.

## Appendix I

# Lower Layer Error Codes

## ACSE Error Codes

Any of the functions that the ACSE-service-user may call (e.g., the **a_*** functions) will return one of the following error codes:

```
#define SUCCESS         0x3000     No Error
```

The following error codes are returned from the ACSE layer:

| | | |
|---|---|---|
| E_ACSE_ENC_ERR | 0x3001 | ACSE Encode error<br>Error in ASN.1 encoding. |
| E_ACSE_SEND_ERR | 0x3002 | ACSE Send Error<br>Error in sending ACSE. |
| E_ACSE_INVALID_CONN_ID | 0x3003 | Invalid connection ID<br>Connection ID is not valid. |
| E_ACSE_INVALID_STATE | 0x3004 | Invalid State<br>ACSE is not in a valid state. |
| E_ACSE_INVALID_PARAM | 0x3005 | Invalid Parameter<br>Parameter sent is invalid. |
| E_ACSE_BUFFER_OVERFLOW | 0x3006 | Buffer Overflow Error<br>ASN.1 buffer overflow. |
| E_ACSE_MEMORY_ALLOC | 0x3007 | Error Allocating Memory<br>Memory Allocation Failed. |

## ACSE Exception Codes

| | | |
|---|---|---|
| EX_ACSE_DECODE | 0x3081 | ACSE Decode Error<br>Can’t decode incoming PDU. |
| EX_ACSE_INVALID_STATE | 0x3082 | Invalid State<br>ACSE is not in the correct state for the received PDU. |

# TP4 Error Codes

| | | |
|---|---|---|
| TP4E_SHMALLOC | 0x1201 | Shared memory error<br>Error in allocating shared memory. |
| TP4E_BADCONN | 0x1202 | Bad Connection State<br>Unable to connect – bad state. |
| TP4E_QUEUE_FULL | 0x1203 | SPDU queue full<br>Session PDU queue is full. |
| TP4E_CONN_STATE | 0x1204 | Illegal Connection state<br>Unable to connect – illegal state. |

The following error codes (TP4E_INVAL_*) indicate that the corresponding parameters in the TP_CFG structure are not legal.

```
#define TP4E_INVAL_TPDU_LEN         0x1205

#define TP4E_INVAL_REM_CDT          0x1206

#define TP4E_INVAL_LOC_CDT          0x1207

#define TP4E_INVAL_SPDU_OUTST       0x1208

#define TP4E_INVAL_NUM_CONNS        0x1209

#define TP4E_INVAL_SPDU_LEN         0x120A

#define TP4E_INVAL_WINDOW_TIME      0x120B

#define TP4E_INVAL_INACT_TIME       0x120C

#define TP4E_INVAL_RETRANS_TIME     0x120D

#define TP4E_INVAL_MAX_TRANS        0x120E

#define TP4E_MALLOC                 0x120F      /* memory allocation
error  */
```

# CLNP Error Codes

The following error codes may be returned from the CLNP API functions:

| | | |
|---|---|---|
| LLC_ERR_SRC_ADDR | 0x0201 | LLC header Source field invalid |
| LLC_ERR_DEST_ADDR | 0x0202 | LLC header Dest field invalid |
| LLC_ERR_CONTROL | 0x0203 | LLC header Control field invalid |

# CLNP General Errors

| | | |
|---|---|---|
| `CLNP_ERR_CFG_FILE` | `0x3400` | Configuration File Errors<br>Errors found in configuration file (or required parameters are not configured. Such as the local MAC and local NSAP. |
| `CLNP_ERR_NOT_INIT` | `0x3401` | CLNP has not been initialized<br>Protocol not started. |
| `CLNP_ERR_MEM_ALLOC` | `0x3402` | Error in allocating memory<br>Cannot allocate memory. |
| `CLNP_ERR_NULL_PTR` | `0x3403` | NULL pointer error<br>Null pointer passed to a **`clnp_function`**. |

# CLNP Parsing clnp_param Structure Errors

| | | |
|---|---|---|
| `CLNP_ERR_NSAP_LEN` | `0x3404` | NSAP length error<br>NSAP length is 0 or more than the allowed value. This is an unrecoverable error during CLNP initialization. |
| `CLNP_ERR_LIFETIME` | `0x3405` | Invalid PDU lifetime<br>Recoverable error during CLNP initialization. Lifetime value will be set to default. |
| `CLNP_ERR_LIFETIME_DEC` | `0x3406` | Invalid PDU lifetime decrement<br>Recoverable error during CLNP initialization. Lifetime decrement value will be set to default. |
| `CLNP_ERR_ESH_CFG_TIMER` | `0x3407` | Invalid ESH Configuration Timer<br>Recoverable error during CLNP initialization. End System Holder timer will be set to default. |
| `CLNP_ERR_ESH_DELAY` | `0x3408` | Invalid Delay Time for First ESH<br>Recoverable error during CLNP initialization. Delay will be set to default. |
| `CLNP_ERR_MAC_ADDR` | `0x3409` | Local MAC address not configured<br>Must have a local MAC address – this is required for ADLC sub-network. Unrecoverable error during CLNP initialization. |
| `CLNP_ERR_UDATA_LEN` | `0x3410` | CLNP-user data length too large<br>User data exceeds maximum length. |

# CLNP PDU Parsing (Decoding) Errors

| | | |
|---|---|---|
| `CLNP_ERR_PDU_MAC_ADDR` | `0x3420` | Error decoding MAC address PDU<br>The NPDU MAC address is not a local MAC or cannot decode ALL End Systems Addresses. |
| `CLNP_ERR_PDU_ID` | `0x3421` | Invalid PDU ID<br>Not a supported PDU. Currently ISO 8473 and ISO 9452 standards are supported. |
| `CLNP_ERR_PDU_VER` | `0x3422` | Invalid PDU version<br>Not a supported PDU version. Currently DT, ER, ESH, and ISH PDUs are supported. |
| `CLNP_ERR_PDU_TYPE` | `0x3423` | Invalid PDU type<br>Not a supported PDU type. Currently DT, ER, ESH, and ISH PDUs are supported. |
| `CLNP_ERR_PDU_LEN` | `0x3424` | Invalid PDU length<br>Received PDU length does not match the length indicated by sub-network. |
| `CLNP_ERR_PDU_EXPIRED` | `0x3425` | PDU expired<br>DT (Data Type) or ER (Error) PDUs lifetime has expired. |
| `CLNP_ERR_PDU_NSAP_ADDR` | `0x3426` | Error NSAP addressing to PDU<br>PDU is improperly addressed to a NSAP that is not assigned locally. |
| `CLNP_ERR_PDU_SEGMENTING` | `0x3427` | Error segmenting PDUs<br>Segmented PDUs are not supported – PDUs must arrive in one packet. |
| `CLNP_ERR_PDU_CHECKSUM` | `0x3428` | Error PDU Checksum<br>PDU checksum verification failed. |
| `CLNP_ERR_PDU_LAST_SEG` | `0x3429` | Segmented PDU Error<br>Last segment bit not set – indicating an unsupported segmented PDU. |
| `CLNP_ERR_PDU_ER_PDU` | `0x342A` | Error ER PDU<br>Code not compiled for ER (Error) PDU processing. |

# Subnetwork API Error Codes

The following error codes may be returned from the Subnetwork API functions:

| | | |
|---|---|---|
| SNET_ERR_INIT | 0x3501 | Error Initializing Sub-network Interface<br>Sub-network interface not available. |
| SNET_ERR_WRITE | 0x3502 | Sub-network Write Function Failed<br>Cannot write to sub-network |
| SNET_ERR_READ | 0x3503 | Sub-network Read Function Failed<br>Cannot read from sub-network or no data to read. |
| SNET_ERR_MAC_INVALID | 0x3504 | Invalid MAC address<br>Unable to obtain requested ALL ES, ALL IS, or local MAC address. |
| SNET_ERR_FRAME_LEN | 0x3505 | Frame Length Error<br>Received more data then reserved in buffer. |
| SNET_ERR_UDATA_LEN | 0x3506 | User Data Length Error<br>Invalid length of data to send (too large) |

The following are Subnetwork errors specific to the Ethernet driver:

| | | |
|---|---|---|
| SNET_ERR_DRV_OPEN | 0x3520 | Open Driver Command Failed<br>Cannot install the **osillc.vxd** (Win95/NT) or **OSILLC$** (Win 3.x) driver. |
| SNET_ERR_DRV_LOC_MAC | 0x3521 | Driver Error for Local MAC Address<br>Failure to obtain local MAC address from the Ethernet board. |
| SNET_ERR_DRV_ADD_ES_ADDR | 0x3522 | ES Address Driver Error<br>Failure to activate All End System Address. |
| SNET_ERR_DRV_BIND_LSAP | 0x3523 | Failure to bind to LSAP<br>Cannot bind **OSILLC$** (Win 3.x driver) to LSAP. |
| SNET_ERR_DRV_POST_BUFS | 0x3524 | Failure to post buffers to driver<br>**OSILLC$** (Win 3.x driver) cannot post buffers. |

## Appendix K

# IEC GOOSE Example Application Framework

This appendix contains information on the IEC GOOSE Example Framework. The application framework is supplied “as is” and is intended to be used as an example. Maintenance of user modifications to this framework are the responsibility of the user.

The IEC GOOSE framework is built on Windows using the IEC GOOSE framework project file (**iecgoose.dsp**) in the main MMS-Lite workspace.

The framework is supplied in the following files:

**iec_comn.c** This file contains common routines for the manipulation of IEC GOOSE pools.

**iec_tx.c** This file contains framework functions for the creation, transmission, and retransmission of IEC GOOSE messages.

**iec_rx.c** This file contains framework functions for the subscription, decoding, and user callback for receiving IEC GOOSE messages.

**iec_demo.h** This file includes the framework definitions (including log masks).

**iec_demo.c** This file drives the framework for initial debug. This file should not be used as part of an overall embedded application.

The framework functions make use of GOOSE API, MMS-EASE Lite, and other framework functions in order to accomplish the requisite work.

A call to the function **demo_init** in **iec_demo.c** is required in order to initialize the demo.

# Framework functions contained within iec_rx.c

The general flow of the framework, for GOOSE reception, is:

- The framework allows the reception of an IEC GOOSE packet (**clnp_snet_read**).
- The packet is checked to see if it is an Ethertype packet.
  If so, the Ethertype header is decoded for further examination. If not, the packet is discarded.
- The packet Ethertype ID is checked to see if it matches with the Ethertype ID being used for IEC GOOSE (currently defined as **ETYPE_TYPE_GOOSE**).

  If the Ethertype ID does not match, the packet is discarded.
- If the Ethertype ID matches, then the subscribed for MAC Addresses are checked (see the function **iecGooseSubscribe**).

  If **GOOSE_DEC_MODE_LAST_RX** is the **decode_method** specified, then there may be only one MAC_Address/GCRef pair. This is due to the fact that only the last received IEC GOOSE message for each MAC address is saved for later decoding. If there is more than one gcRef for the same MAC, then some GOOSE messages would not be saved.

If **GOOSE_DEC_MODE_IMMEDIATE** is the **decode_mode** specified, then there may be multiple GCRefs associated with a single MAC_Address.

- If the **decode_mode** is **GOOSE_DEC_MODE_IMMEDIATE**, the decode function **iecGooseDecode** is called.

  The decode function finds the appropriate MAC/GCRef combination based upon a header decode.

  If there is a **stNum** change detected, the decode continues.

  The subscribed for the received **DataEntry** list is then scanned to see if any of the information received is to be delivered to the application (based upon the function **gse_iec_data_init**).

  The databuffer is then marked as **GOOSE_CALLBACK_REASON_STATECHANGE_DATA_UPDATED**. This allows the application to determine which buffers have been updated.

  **Note**: It is a general philosophy of the framework that the DataSet being published may be a superset of the information needed by the application. Therefore, the subscription process allows a subset of the published information to be subscribed for.

  The user callback function is called indicating the appropriate status and information.

- If the **decode_mode** is **GOOSE_DEC_MODE_LAST_RX**, then the application must call the function **iecGooseLastRxDecode** in order to decode the last received GOOSE for each MAC address.

  **Note**: This decode mode may improve performance (less CPU time) for implementations that desire to have the GOOSE information be synchronized with the internal Input I/O scan.

  The maximum number of GEESE that can be received is specified by **MAX_RXD_GOOSE**.

## iecGooseSubscribe

**Usage:** This function is used by a framework user to subscribe for an IEC GOOSE.

**Function Prototype:**

```
IEC_GOOSE_SEND_USER_INFO  *iecGooseSubscribe (ST_UCHAR *DstAddress,
                                              ST_CHAR *gcRef,
                                              ST_CHAR *DataSetRef,
                                              ST_CHAR *AppID,
                                              ST_INT ConfRevNum,
                                              ST_INT numDataEntries,
                                              ST_TYPE_ARRAY *rt_array,
               ST_VOID (*usr_fun)(IEC_GOOSE_SEND_USER INFO *info,
                                              GSE_IEC_CTRL *gptr,
                                              ST_VOID *usr,
                                              ST_UINT16 reason),
                                              ST_INT decode_mode);
```

**Parameters:**

`DstAddress` A pointer to a buffer that contains the six byte MAC Address to which the expected IEC GOOSE message is being sent. This parameter is used to configure the MAC filtering. The buffer need not be persistent.

`gcRef` A pointer to a buffer that contains the GOOSE Control Block Reference that is to be expected. The buffer must be persistent. If this value does not match with the value supplied by the received GOOSE, a error will be indicated and no further processing of that GOOSE packet will occur (see the static function **`iecGooseDecode`** if this check needs to be removed).

`DataSetRef` A pointer to a buffer that contains the Data Set Reference that is to be expected. The buffer must be persistent. If this value does not match with the value supplied by the received GOOSE, a error will be indicated and no further processing of that GOOSE packet will occur (see the static function **`iecGooseDecode`** if this check needs to be removed).

`AppID` A pointer to a buffer that contains the Application ID that is to be expected. The buffer must be persistent. If this value does not match with the value supplied by the received GOOSE, a error will be indicated and no further processing of that GOOSE packet will occur (see the static function **`iecGooseDecode`** if this check needs to be removed).

`ConfRevNum` This value represents the Configuration Revision Number that is to be expected. If this value does not match with the value supplied by the received GOOSE, a error will be indicated and no further processing of that GOOSE packet will occur (see the static function **`iecGooseDecode`** if this check needs to be removed).

## iecGooseSubscribe (cont'd)

**Parameters (cont'd):**

`numDataEntries` This specifies the number of Data Entries to be configured. This value determines the size of the **`rt_array`**.

`rt_array` Is an array (size of **`numDataEntries`**) of **`RT_TYPE_ARRAY`**:

```
typedef struct rt_type_array{
      ST_INT num_rts;    //number of runtime type elements
                         //that define the actual data type for the entry
      RUNTIME_TYPE *rt;  //pointer to the head of the runtime type .
      }RT_TYPE_ARRAY;
```

`usr_fun` This is the callback function desired to be called when an incoming GOOSE that matches all filter criteria (e.g., **`DstAddress`**, **`gcRef`**, etc.) is met.

The callback is supplied the following information:

`GOOSE_SEND_USER INFO *` the handle created for this subscription

`GSE_IEC_CTRL *` pointer to the GOOSE control created.

`ST_VOID *` user information stored after creation in the handle structure.

`ST_UINT16` reason for callback. The set of defined reasons may be found in **iec_goose_sample.h**, but include the following:

```
GOOSE_CALLBACK_REASON_STATECHANGE_DATA_UPDATED
GOOSE_CALLBACK_REASON_TIMEALLOWED_TO_LIVE_TIMEOUT
GOOSE_CALLBACK_REASON_OUT_OF_SEQUENCE_DETECTED
GOOSE_CALLBACK_REASON_CONFREV_MISMATCH
GOOSE_CALLBACK_REASON_NEED_COMMISSIONING
GOOSE_CALLBACK_REASON_TEST_MODE
GOOSE_CALLBACK_REASON_GCREF_MISMATCH
GOOSE_CALLBACK_REASON_APPID_MISMATCH
GOOSE_CALLBACK_REASON_DATSET_MISMATCH
```

The values are or'd together to form a reason mask.

`decode_mode` This parameter specifies the processing directive for a GOOSE that matches the filter criteria. The allowed values are:

`GOOSE_DEC_MODE_LAST_RX` decode occurs by application calling the function **iecGooseLastRxDecode**.

`GOOSE_DEC_MODE_IMMEDIATE` decode occurs immediately when message is received.

**Return Value:**

`IEC_GOOSE_SEND_USER_INFO  *` A handle to the user info.
`NULL` Error occurred in subscription.

## iecGooseSubscribe (cont'd)

**Comments:** This function makes calls to the following GOOSE API functions:

| | |
|---|---|
| **`gse_iec_control_create`** | Creates a GOOSE control block |
| **`gse_iec_data_init`** | Creates Data Entries that can be searched and also the storage for the decoded data buffers. |
| **`gse_set_multicast_filters`** | Sets the MAC filtering within the driver. |

## iecGooseUnSubscribe

**Usage:** This function destroys resources allocated with a created IEC GOOSE subscription.

**Function Prototype:**

```
ST_RET iecGooseUnSubscribe(IEC_GOOSE_SEND_USER_INFO *goosehandle);
```

**Parameters:**

| | |
|---|---|
| `goosehandle` | The handle value returned by the function **iecGooseSubscribe**. |

| | | | |
|---|---|---|---|
| **Return Value:** | `ST_RET` | `SD_SUCCESS` | IEC GOOSE sent successfully. |
| | | `<>0` | Error code. |

**Comments:** This function makes calls to the following GOOSE API functions:

| | |
|---|---|
| **`gse_iec_control_destroy`** | Destroys a GOOSE control block. |
| **`gse_set_multicast_filters`** | Sets the MAC filtering within the driver. |

## iecGooseLastRxDecode

**Usage:** This function is used to drive the decodes of received GEESE that were subscribed to as `GOOSE_DEC_MODE_LAST_RX` (see the function **`iecGooseSubscribe`**).

**Function Prototype:** `ST_RET iecGooseLastRxDecode (ST_VOID);`

**Parameters:** None

**Return Value:** `SD_SUCCESS` if the GOOSE decoding was successful; otherwise `SD_FAILURE`.

**Comments:** This function searches for a LAST_RX subscription and then calls the function **`iecGooseDecode`**. This function makes GOOSE API calls to:

| | |
|---|---|
| **`gse_iec_hdr_decode`** | Decodes the IEC GOOSE header |
| **`ms_asn1_to_local`** | Converts GOOSE data into local memory representation. |

## get_goose_messages

**Usage:** This function is used to receive IEC GOOSE messages from the driver, It then process them according to the decode mode specified by the function **`iecGooseSubscribe`**.

**Function Prototype:** `ST_RET get_goose_messages(ST_VOID);`

**Parameters:** None

**Return Value:** `SD_SUCCESS` if a packet was received from the driver; otherwise `SD_FAILURE`.

**Comments:** This function makes calls to the following GOOSE API functions:

| | |
|---|---|
| **`clnp_snet_read`** | Obtains a GOOSE packet |
| **`chk_for_goose_msg`** | This function is where detection of the destination Ethertype ID occurs. It will need to be modified in order to extend the framework to support SampledValues, GSE Management, GSSE (formerly UCA GOOSE), and other link level messages. |

## chk_for_goose_msg

**Usage:** This function is called in order to process GOOSE, GSSE, or GSE Management functions.

**Function Prototype:**

```
ST_RET chk_for_goose_msg (ST_UCHAR *loc_mac,
                          ST_UCHAR *rem_mac,
                          ST_INT pdu_len,
                          ST_CHAR *pdu);
```

**Parameters:**

| | |
|---|---|
| `loc_mac` | Pointer to the destination MAC. This MAC address should be the local unicast address. |
| `rem_mac` | Pointer to the source MAC of the sending node. |
| `pdu_len` | This is the length of the Link Protocol Data Unit (LPDU) to be processed. This length should be the Virtual LAN Type ID (0x8100) which signals an Ethertype frame. The actual length of the LPDU can be obtained with a call to the function **`etype_hdr_decode`**. |
| `pdu` | This is a pointer to the LPDU data buffer minus the two MAC addresses. |

**Return Value:** `SD_SUCCESS` if a packet was processed; otherwise `SD_FAILURE`.

## chk_iec_goose_timeout

**Usage:** This function is used to detect when a rx'd/subscribed GOOSE has an expired TAL (Time Allowed to Live).

**Function Prototype:**

```
ST_VOID chk_iec_goose_timeout (ST_INT32 elapsed_msec);
```

**Parameters:**

`elapsed_msec` Value of elapsed time in msec since function was called last.

**Return Value:** None.

# Framework functions contained within iec_tx.c

## iecGoosePubCreate

**Usage:** This function is used by a framework user to create publishing GEESE.

**Function Prototype:**

```
IEC_GOOSE_SEND_USER_INFO  *iecGoosePubCreate (ST_CHAR *gcRef,
                                               ST_CHAR *DataSetRef,
                                               ST_CHAR *AppID,
                                               ST_ULONG ConfRevNum,
                                               ST_BOOLEAN NeedsComm,
                                               ST_UINT16 tci,
                                               ST_UINT16 etypeID
                                               ST_UINT16 appID);
```

**Parameters:**

| | |
|---|---|
| `gcRef` | A pointer to a buffer that contains the GOOSE Control Block Reference that is to the value to be sent in the IEC GOOSE. The referenced control need not exist locally. The value must be persistent. |
| `DataSetRef` | A pointer to a buffer that contains the DataSet Reference that is to be the value to be sent in the IEC GOOSE. The referenced DataSet must be defined and present within the server. The value must be persistent. |
| `AppID` | A pointer to a buffer that contains the Application ID that is to be sent in the IEC GOOSE. The buffer must be persistent |
| `ConfRevNum` | This value represents the Configuration Revision Number that is to be sent. |
| `NeedComm` | This flag represents the value of the IEC GOOSE NeedsCommissioning parameter. A value of `SD_TRUE` indicates that commissioning is required. |
| `tci` | The value of the Virtual LAN’s Tag Control Information. These values are predefined in **ethertyp.h**. |
| `etypeID` | The value of the Ethertype ID, as defined in **ethertyp.h**. |
| `appID` | The value of the Application Identifier. If no **`appID`** is configured in the application, the default value of 0x0000 should be passed into this function. |

| **Return Value:** | | |
|---|---|---|
| | `IEC_GOOSE_SEND_USER_INFO *` | A handle to the user info. |
| | `NULL` | Error occurred. |

**Comments:** This function makes calls to the following GOOSE API functions:
**`gse_iec_control_create`** Creates a GOOSE control block.
**`Gse_iec_data_init`** Creates Data Entries that can be searched and the storage for the decoded data buffers.

## iecGoosePubDestroy

**Usage:** This function destroys resources allocated with a created IEC GOOSE publication.

**Function Prototype:**

```
ST_RET iecGoosePubDestroy (IEC_GOOSE_SEND_USER_INFO *goosehandle);
```

**Parameters:**

| | |
|---|---|
| `goosehandle` | The handle value of type **`IEC_GOOSE_SEND_USER_INFO`** returned by the function **`iecGoosePubCreate`**. |

| | | | |
|---|---|---|---|
| **Return Value:** | `ST_RET` | `SD_SUCCESS` | This function always returns `SD_SUCCESS`. |

**Comments:** This function makes calls to the following GOOSE API function:

| | |
|---|---|
| **`gse_iec_control_destroy`** | Destroys a GOOSE control block |

## iecGoosePublish

**Usage:** This function polls for the GOOSE data by invoking the MMS-EASE Lite **`read_ind`** functions. After the poll is complete, it updates the data in the GOOSE Control Data Entries and then starts the sequence of transmission.

**Function Prototype:**

```
ST_VOID iecGoosePublish (IEC_GOOSE_SEND_USER_INFO *goosehandle);
```

**Parameters:**

| | |
|---|---|
| `goosehandle` | The handle value of type **`IEC_GOOSE_SEND_USER_INFO`** returned by the function **`iecGoosePubCreate`**. |

**Return Value:** None.

**Comments:** This function calls the following internal framework functions:

| | |
|---|---|
| **`mvlu_rpt_scan_read`** | Polls for the data (from the MMS-LITE API). |
| **`gse_iec_data_update`** | Updates the GOOSE information with the polled data. |
| **`gse_iec_encode`** | Encodes the GOOSE. |
| **`gse_iec_send`** | Sends the GOOSE. |

## start_trans_goose

**Usage:** This function is used to encode/transmit a state changed GOOSE. It increments the **stNum** and **sqNum**. It does not change the event timestamp (This must be set by the application).

**Function Prototype:**

```
ST_RET start_trans_goose (GSE_IEC_CTRL *gptr, RETRANS_CURVE *retrans_curve);
```

**Parameters:**

gptr The handle value of type **GSE_IEC_CTRL** returned by the function **gse_iec_control_create**.

RETRANS_CURVE Pointer of type **RETRANS_CURVE** to the retransmission curve specification.

```
typedef struct retrans_curve{
   ST_UINT  num_retrans;        //number of active entries in array
   ST_UINT32retrans[MAX_NUM_RETRANS]; //msec retrans
}RETRANS_CURVE;
```

| | | | |
|---|---|---|---|
| **Return Value:** | ST_RET | SD_SUCCESS | IEC GOOSE sent successfully. |
| | | <>0 | Error |

**Comments:** This function makes calls to the following GOOSE API functions:

| | |
|---|---|
| **gse_iec_encode** | Encodes the GOOSE |
| **gse_iec_send** | Sends the GOOSE. |

## retrans_goose

**Usage:** This function is used to detect when a GOOSE needs to be retransmitted.

**Function Prototype:**

```
ST_VOID retrans_goose (ST_INT32 elapsed_msec);
```

**Parameters:**

| | |
|---|---|
| `elapsed_msec` | Value of elapsed time (in msec) since function was called last. |

**Return Value:** None.

**Comments:** This function makes calls to the following GOOSE API functions:

| | |
|---|---|
| **`gse_iec_encode`** | Encodes the GOOSE. |
| **`gse_iec_send`** | Sends the GOOSE. |