MEMO

SINTEF ICT

Address:	NO-7465 Trondheim, NORWAY
Location:	S P Andersens v 15 NO-7031 Trondheim
Telephone:	+47 73 59 30 00
Fax:	+47 73 59 43 02

Enterprise No.: NO 948 007 029 MVA

MEMO CONCERNS

A Description of the Manufacturing Message Specification (MMS)

DISTRIBUTION	FOR YOUR ATTENTION	COMMENTS ARE INVITED	FOR YOUR INFORMATION	AS AGREED
Stig Ole Johnsen, SINTEF T&S			x	
Kai Hansen, ABB				x

FILE CODE	CLASSIFICATION
	Public

ELECTRONIC FILE CODE

MMS_Notat.doc

PROJECT NO.	DATE	PERSON RESPONSIBLE / AUTHOR	NUMBER OF PAGES
504064.20	2007-08-06	Jan Tore Sørensen, Martin Gilje Jaatun	46

Table of contents

1 Introduction

This memo describes the Manufacturing Message Specification (MMS), a protocol used in industrial networks. We will focus on the basic architecture and functions of MMS, and wish to provide the reader with relevant background information needed to understand the protocol. MMS is declared an international standard, named ISO 9506, and is currently developed and maintained by the ISO Technical Committee 184 (TC184). For a more detailed description we refer to the standard [1] and a collection of white-papers from SISCO [2] [3, 4].

2 Manufacturing Message Specification (MMS) Basics

MMS is an application layer protocol which specifies services for exchange of real-time data and supervisory control information between networked devices and/or computer applications. It is designed to provide a generic messaging system for communication between heterogeneous industrial devices. The specification only describes the network visible aspects of communication. By choosing this strategy, the MMS does not specify the internal workings of an entity, only the communication between a client and a server, allowing vendors full flexibility in their implementation. In order to provide this independence, the MMS defines a complete communication mechanism between entities, composed of [3]:

1. **Objects:** A set of standard objects which must exist in every conformant device, on which operations can be executed (examples: read and write local variables, signal events).
2. **Messages:** A set of standard messages exchanged between a client and a server station for the purpose of controlling these objects
3. **Encoding Rules:** A set of encoding rules for these messages (how values and parameters are mapped to bits and bytes when transmitted)
4. **Protocol:** A set of protocols (rules for exchanging messages between devices).

MMS composes a model from the definition of objects, services and behaviour named the Virtual Manufacturing Device (VMD) Model. The VMD model uses an object oriented approach to represent different physical industrial (real) devices in a generic manner. Some of these objects are variables, variable type definitions, programs, events, historical logs (called journals) and semaphores. Along with the definition of these objects, MMS defines a set of communications services that an application can use to manipulate these objects.

We observe that in the literature the terms services, service primitives and messages are all used to describe the functions that manipulate objects or their attributes. We will therefore in this memo use the term service primitive as this is used in the ISO 9506 standard, unless we are citing directly from a written source, in which case the quote will be evident in the text. The standard also refers to physical industrial devices as “real devices” and we will continue to use this terminology to avoid confusion.

As MMS is based on an object oriented approach, we will give a brief introduction to the addressing and object hierarchy of MMS, before focusing on the network communication.

2.1 Architecture and Addressing

The MMS architecture is based on a common client-server model. Real devices used in industrial networks often contain an MMS server allowing the device to be monitored and managed from an MMS client. An MMS client is typically part of a Control Builder application, Human -Machine Interface (HMI) or an MMS to OLE for Process Control (OPC) gateway (MMS/OPC GW). The ABB Control Builder is an application used to program and monitor industrial controllers such as ABB's AC 800 M. Both the control builder and the MMS/OPC GW uses service primitives provided in the MMS to manage devices containing MMS servers. This is depicted in Figure 1.

As MMS does not specify how to address clients and servers, an entity containing an MMS client or server must rely on the addressing scheme of underlying protocols in the process of establishing an application association to support the MMS environment [1]. In practice, clients and servers are addressed by their IP address and the MMS server uses TCP port number 102. The addressing allows for an MMS context to be negotiated between two peer applications.

To address an MMS object variable, MMS provides several different address modes. MMS allows an address to have different syntax, based on the implementer's choice of what is most appropriate for that device. The specification separates between named and unnamed variables. The unnamed variables are identified by a fixed physical address in the VMD, expressed by either:

- **Numeric:** A numeric address is represented by an unsigned integer number (e.g., `Unsigned32 173`).
- **Symbolic:** A symbolic address is represented by a character string (e.g., `VisibleString "C076"`).
- **Unconstrained:** An unconstrained address is represented by a untyped string of bytes (e.g., An `OCTET STRING "0x57AB"`).

Named variables are identified by an object name (e.g., a string of characters), which is VMD specific, domain specific or Association-specific. An MMS server may declare an MMS object variable that does have a specific address, but choose not to reveal this address to MMS clients. If this is the case the object variable shall be defined locally and with a specific access method other than public [1]. Access control in MMS is enforced through the use of access control list objects containing access methods for objects and appurtenant object variables. Once an MMS communication context is established between a client and a server, the standard specifies details for the MMS objects, variables, object hierarchy, and service primitives.

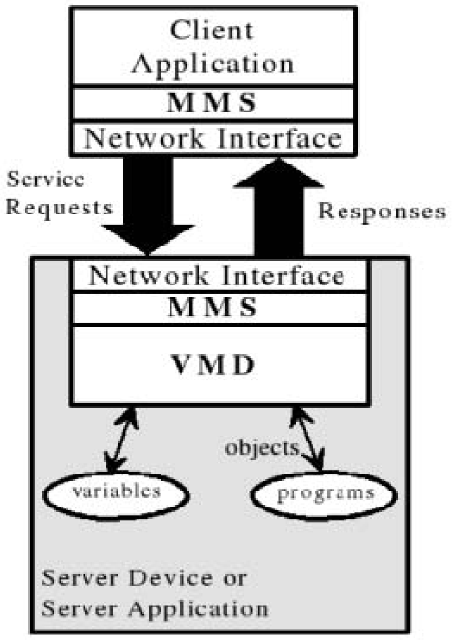


Figure 1: VMD communication model between control builder and MMS client [2]

2.2 MMS Objects, Services Primitives and Access Control

Associated with each object is a set of variables that describe values in a given instance of the object. For each object there are corresponding MMS service primitives that allow client applications to access and manipulate those objects. The top level object in the MMS is the VMD which has at least one network-visible address.

Each real device is represented by a real object with vendor specific features associated with them. The VMD model maps the real object and devices onto virtual objects and devices, described in a generic manner which is in conformance to the VMD model. In other words a real variable is an element of typed data that is contained within a VMD object. An MMS variable is part of a virtual object that represents a mechanism for the MMS client to access the real variable. The MMS server containing the virtual MMS object can be understood as a communication driver which hides the specifics of a real device from the client. From the client's point of view the virtual MMS variable represents a pointer or an access method to the real variable and it is only the MMS server with its objects and its behaviour that is visible to the client. The MMS client can never interact with real device variables directly.

All MMS objects contain an access method variable. This attribute contains the information which a device needs to identify the real variable as described above. It contains values which are necessary to find the memory location of the real variable with the contents that lie outside MMS. A special method, the method PUBLIC, is standardized for accessing the real variables.

A table listing all MMS objects with a short description can be found in Appendix A.

For each object there are corresponding MMS service primitives that allow client applications to access and manipulate those objects. The MMS defines the service primitives of both clients and servers, but the VMD focuses only on specifying the network visible behaviour of MMS servers. And thus, each vendor of an MMS server device is responsible for hiding vendor specific details of the real objects and devices by providing an executive function which maps the real entities up to the virtual level, which shall comply with the VMD model definitions. To ensure vendor implementation compliance with the VMD model, it specifies how MMS devices containing a MMS server shall provide a consistent and well defined view of the object contained in the VMD. And thus, MMS provides a common interface for communication with different devices through the generic virtual objects.

All MMS objects listed in Table 4, except the Operator Station object, inherit six abstract services from the VMD object. These are depicted and described in Table 1. E.g. service primitives `read` and `RequestDomainUpload` for the objects **Named Variable List** and **Domain** respectively inherit from the abstract service primitive `get`.

MMS General methods	Description
Get	This method is used to obtain the value of a specified object.
Set	This method is used to write/put value or contents into a specified object.
Query Attributes	This method is used to obtain structure or capability information of a specified object.
Create	This method allows objects of particular classes to be instantiated.
Rename	This method allows instantiated objects to be renamed.
Delete	This method allows instantiated objects to be destroyed.

Table 1: The basic methods (services) inherited from the VMD object [3].

MMS uses access control lists to provide explicit control of the ability to access or alter MMS objects. Protection requirements for an MMS variable are inherited from the underlying real variable in the real device. These requirements are established by the access method in the MMS object. ISO 9506[1] states that each object within an MMS implementation must contain a reference to an Access Control List object that specifies the conditions under which services directed at the named object may succeed. For the purposes of specifying the control conditions, services are grouped into six classes as described in Table 1. Access control is enforced through special mechanisms provided by MMS. These mechanisms include possession of a semaphore, identity of user (Application Reference), and the submission of a password (which may be arbitrarily complex).

2.3 Network Services

As we have stated earlier MMS is not by itself a communication protocol, as it only defines messages that have to be transported by an unspecified network. MMS was originally developed as a part of the Manufacturing Automation Protocol (MAP) specification and is therefore specified on all seven OSI layers as depicted in Figure 3. MAP was originally created by General Motors as an internal standard for communications in industrial automation networks. It is now a public, multivendor communications standard for industrial automation equipment. For more information about MAP we refer to [5]. MMS supports the use of both confirmed and unconfirmed services, but we will in this memo focus on the confirmed services. The MMS defines the following PDUs for a confirmed service exchange:

1. Confirmed-RequestPDU
2. Confirmed-ResponsePDU
3. Confirmed-ErrorPDU
4. Cancel-RequestPDU
5. Cancel-ResponsePDU
6. Cancel-ErrorPDU
7. RejectPDU

These messages will be used in the communication between a client and a server when a client wishes to invoke a service primitive. A normal MMS request between client and server follows the pattern depicted in Figure 2 using the enumerated list above.

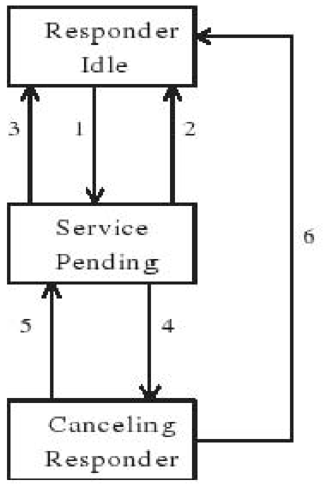


Figure 2: A Confirmed Service state machine as seen by the MMS server

Before a service primitive is called through a Confirmed-RequestPDU, the server is in a Responder Idle state as seen Figure 2. Upon receipt of a Confirmed-RequestPDU for any of the confirmed services, the MMS-provider issues an indication primitive specifying the particular service being requested and an invoke ID that specifies the service instance and enters the state Service Pending. Upon receipt of a response service primitive containing a result parameter specifying the service previously indicated and an invoke ID that specifies the service instance, the MMS-provider sends a Confirmed-ResponsePDU which specifies the service type and the invoke ID from the response primitive. Then a state transition into the Responder Idle state occurs. Upon receipt of a response service primitive containing a Result parameter specifying the service previously indicated and an invoke ID that specifies the service instance, the MMS-provider sends a Confirmed-ErrorPDU specifying the service type and the invoke ID from the response primitive. A state transition into the Responder Idle state then occurs. Upon receipt of a Cancel-RequestPDU specifying the invoke ID of the matching service instance, the MMS-provider issues a cancel indication service primitive specifying the invoke ID of the service request to be canceled this information is obtained from the

Cancel-RequestPDU parameters. The state Canceling Responder is then entered. The RejectPDU is issued if the responder receives a malformed PDU.

According to [1], the MMS runs on the network stack depicted in Figure 3. As all ISO standards this network stack relates to the Open System Interconnection stack describing the abstract service layers such as session and presentation layer. We will now give a short description of some of the relevant protocols/layers

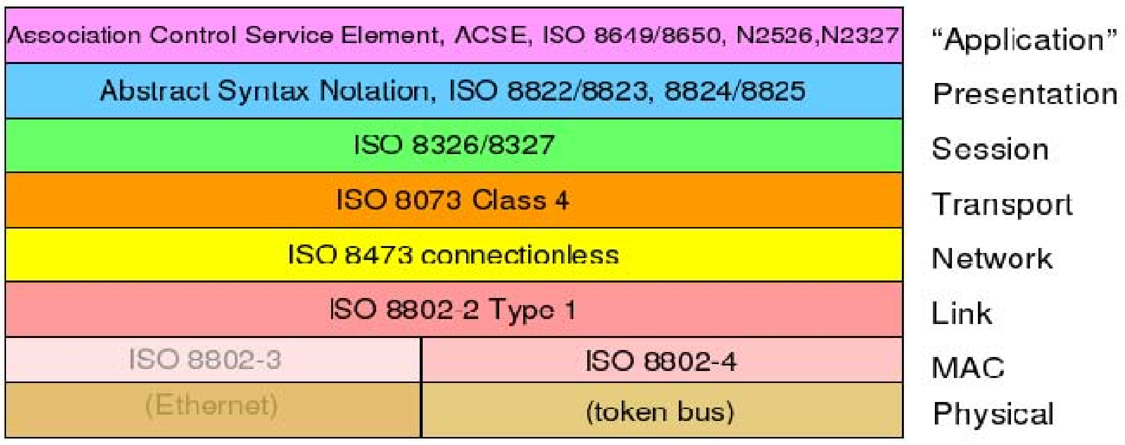


Figure 3: The MMS communication stack

2.4 Application Layer: ACSE

On top of the stack we find ISO's Association Control Service Element (ACSE) protocol. This protocol is specified in ISO documents [6, 7]. ACSE is used to establish and release Application Associations (AA) between Application Entity (AE) and to determine the identity and application context of that association. An application-association is defined by [6] as the cooperative relationship between two AEs. This relationship provides the necessary frame of reference between the AEs so that they may interwork effectively. An application context is an explicitly identified set of application-service-elements, related options and any other necessary information for the interworking of application entities in an application association.

ACSE supports two modes of communication service: connection-oriented and connectionless. The ACSE connection-oriented service is provided through the connection-oriented mode of the ACSE protocol in conjunction with the connection-oriented presentation layer service [7]. Likewise, for the ACSE connectionless service is provided through the connectionless mode of the ACSE protocol in conjunction with the connectionless presentation-service.

For the connectionless service the application-association (AA) exists only during the invocation of the single ACSE connectionless mode service, named A-UNIT-DATA. The mappings of the ACSE services to presentation services and ACSE APDUs are shown in Table 2.

ACSE service	**APDU**	**Communication mode**	**Service type**
A-ASSOCIATE	AARQ, AARE	Connection oriented	confirmed
A-RELEASE	RLRQ, RLRE	Connection oriented	confirmed
A-ABORT	ABRT	Connection oriented	non-confirmed
A-P-ABORT	ABRT	Connection oriented	Provider initiated
A-UNIT-DATA	ADAT	Connectionless	non-confirmed

Table 2: Mapping of ACSE primitives to the ISO presentation layer services [7]

The functionality of an AE is factored into a number of application-service-elements (ASEs). The interaction between AEs is described in terms of the use of their ASE's services [8]. Three functional units are defined for the connection oriented services in ACSE. Each of the functional units uses the services in Table 2 and communicates through the APDUs. Each APDU contains a set of variables, through which information are mediated between the AEs. The mandatory Kernel functional unit is used to establish and release basic application-associations. For enhanced functionality the ACSE includes two optional functional units. The Authentication functional unit supports the exchange of information in support of authentication during association establishment. It provides additional facilities for exchanging information in support of authentication during association establishment without adding services.

The ACSE authentication facilities may be used to support a limited class of authentication methods, but as we will discuss in section 3.3, none of these methods seem to be implemented in SISCO's MMS implementation. These methods are:

- Credentials
- Passwords
- Peer-entity authentication

The second optional functional unit supports the negotiation of application context during association establishment. The connectionless mode of ACSE does not have the notion of functional units. Nor does it support authentication as the connection-oriented mode of ACSE. As stated in Table 2, ACSE has a number of services which supports the creation of application-association and application context. We conclude our summary of the ACSE by giving a short description of each of the services.

A-ASSOCIATE: This confirmed service is used to initiate an application association between application entities through the Application Context Name parameter.
A-RELEASE: This confirmed service is used to release an application association between application entities without loss of information.
A-ABORT: This unconfirmed service causes the abnormal release of an association with a possible loss of information.
A-P-ABORT: This provider-initiated service indicates the abnormal release of an application association by the underlying presentation service with a possible loss of information.
A-UNIT-DATA: This connectionless service simultaneously establishes and releases an association.

2.5 Presentation Layer: ASN.1

The presentation layer protocol is used to transmit information between open systems using connection oriented or connectionless mode[1]transmission at the presentation layer of the OSI 7 layer model [9]. The presentation layer exists to ensure that the information content of presentation data values is preserved during transfer and to add structure to the units of data that are exchanged. It has two functions it carries out on behalf of the upper layers:

- Negotiation of transfer syntax.
- Transformation of syntax.

The transfer syntax negotiation allows presentation protocols to agree on one transfer syntax. The other task of the presentation layer is to transform the application layer abstract syntax into transfer syntax and vice versa.

A set of presentation data value definitions associated with an application protocol constitutes an abstract syntax. The abstract syntax specification identifies the information content of a given set of presentation data values. It does not identify the transfer syntax to be used while presentation data values are transferred between presentation entities, nor is it concerned with the local representation of presentation data values [10]. It is the responsibility of cooperating application entities to determine the set of abstract syntax they employ in their communication and inform the presentation entities of this agreement. Knowing the set of abstract syntax to be used by the application entities, the presentation entities are responsible for selecting mutually acceptable transfer syntax that preserve the information content and structure of the presentation data values [11].

Presentation entities support protocols that enhance the OSI session service in order to provide a presentation service. The role of the presentation layer is to provide a representation of presentation data values in the User data parameters of session service primitives.

In OSI, ASN.1 is the description language used to define data types [12]. ASN.1 is based on the Backus system and uses the formal syntax language and grammar of the Bachus-Naur Form (BNF)[13]. As seen in Figure 3, MMS uses ASN.1 as abstract syntax notation at the presentation layer. An abstract syntax notation is the notation used in defining data structures or set of values for messages and applications. The abstract syntax notation is then encoded with a set of encoding rules before transmission. These rules transform any message defined using the abstract syntax notation, in such a way that the actual bits on the transmission medium carry the semantics of the message to the receiver. We call such rules encoding rules, and we say that the result of applying them to a set of abstract syntax notation defined messages for a given application defines the transfer syntax for that application [14]. A collection of ASN.1 definitions relating to a common theme (e.g., a protocol specification) is termed an ASN.1 module. Each module is constructed using the same high level syntax:

[1] For connectionless mode transmission, the sending presentation entity selects the transfer syntax. No transfer syntax negotiation occurs.

```
<module> DEFINITIONS ::=

BEGIN

<linkage>

<declarations>

END
```

The <module> term names the module. It consists of two parts:

- **A module name:** A name that provide a textual description of the module.
- **An (optional) object identifier:** An identifier that provide an authoritative name for the module. This name unambiguously distinguishes the module from all other defined ASN.1 modules.

The <linkage> term relates this module with other modules. It may include object definition imports from other modules and may define which object definitions this module wishes to export. The last part of the module, the <declaration> term, contains the actual ASN.1 object definitions. There are three kinds of objects defined using ASN.1; type, value and macros. All ASN.1 objects are named using an alphabetic case convention to indicate the kind object the name refers to:

- *Type,* the name starts with an uppercase letter, (e..g, Ipaddress). This convention applies to both simple types and structured types.
- *Value,* the name starts with a lowercase letter (e.g., ipNetToMediaPhysAddress).
- *Macro,* the name consists entirely of uppercase letters (e.g., OBJECT IDENTITY).

ASN.1 types are defined using the syntax below:

```
NameOfType ::= TYPE
```

ASN.1 distinguishes between two kinds of data types; simple and structured. [12] defines a set of simple data types (e.g., INTEGER, BOOLEAN) and provide a facility to construct new elements, structured data types[2], with their own typing inherited in the structure. This allows new data types to be defined which are uniquely identifiable within an application. The structured types may have optional components, possibly with default values. Structured data types use the keywords listed below to construct such data types.

- SEQUENCE – an ordered collection of one or more types.
- SEQUENCE OF – an ordered collection of zero or more occurrences of a given type.
- SET – an unordered collection of one or more types.
- SET OF – an unordered collection of zero or more occurrences of a given type.

To eliminate ambiguity and provide unique identification of data types when they are transmitted over the network, ASN.1 associates each data type with a tag. This tag is a handle that identifies a particular ASN.1 data type. Tagging is also used to distinguish

[2] Structured data types are sometimes referred to a constructor types.

component types within a structured type. For instance, optional components of a SET or SEQUENCE type are typically given distinct context-specific tags to avoid ambiguity. Other keywords which are relevant for this memo are CHOICE, ANY, OPTIONAL and DEFAULT. We include a short explanation of these keywords below.

The CHOICE type denotes a union of one or more type alternatives. The ANY type denotes an arbitrary value of an arbitrary type, where the arbitrary type is possibly defined in the definition of an object identifier or integer value. The keyword DEFAULT indicates that the value of a component is optional, and assigns a default value to the component when the component is absent. Objects with default values, are assumed to use the default value unless another value is transmitted. As we shall see in section 2.6 default values are not encoded in Basic Encoding Rules (BER) for transmission.

OPTIONAL types in module definition are optional to use and are not encoded to BER unless used.

A tag consists of two parts, a class, which is an optional class name, and a number, which is the tag number within the class represented as a non-negative integer. The tags are classified into four classes, based on different requirements on the tags:

1. Universal: Available for use within any protocol. The primitive data-types INTEGER, OCTET string, OBJECT IDENTIFIER, and NULL, are universal. The basic constructors, such as SEQUENCE, also are universal.
2. Application: Available within a specific application. For example, the IpAddress data-types is available for use throughout the TCP/IP network management application.
3. Context specific: This data-type is contained in a larger data-type. The identifier has a unique meaning within the context of the larger data-type. The Context specific data type is used in the MMS protocol.
4. Private: Included so that ASN.1 could be used by private organizations to define proprietary data-types.

There are two ways to tag a type: implicitly and explicitly.

Implicitly tagged types are derived from other types by changing the tag of the underlying type. Implicit tagging is denoted by the ASN.1 keywords `{{class}{number}}IMPLICIT`.

Explicitly tagged types are derived from other types by adding an outer tag to the underlying type. In effect, explicitly tagged types are structured types consisting of one component, the underlying type. Explicit tagging is denoted by the ASN.1 keywords `{{class}{number}}EXPLICIT`.

The tag `{{class}{number}}` alone is the same as explicit tagging, except when the module in which the ASN.1 type is defined, has implicit tagging by default.

For purposes of encoding, an implicitly tagged type is considered the same as the underlying type, except that the tag is different. An explicitly tagged type is considered like a structured type with one component, the underlying type. Implicit tags result in shorter encodings, but explicit tags may be necessary to avoid ambiguity if the tag of the underlying type is indeterminate (e.g., the underlying type is CHOICE or ANY).

ASN.1 uses the concept of values to define an instance of a type. These values are often referred to as abstract values to emphasize that we are considering them without any concern for how they might be represented in a computer or on a communication medium. ASN.1 values are defined using the syntax below:

```
nameOfValue NameOfType ::= VALUE
```

Finally, ASN.1 uses macros to allow users to define additional semantic information. Macros allow the ASN.1 grammar to be extended to meet the future needs of the abstract syntax designer. ASN.1 macros are defined using the syntax below:

```
<macro> MACRO ::=
BEGIN
TYPE NOTATION ::= <type syntax>
VALUE NOTATION ::= <value syntax>
<supporting syntax>
END
```

The <macro> term names the macro, the <type syntax> defines the grammar rules which follows later in the macro definition. Each instance of the macro has a value associated with it, the <value syntax> defines the possible values that the macro instance may take on. The <supporting syntax> provides any additional grammar rules for either the macro’s type or value notation.

The ASN.1 representation of data is independent of machine-oriented structures and encodings and also of the physical representation of the data (referred to as transfer syntax in communication terminology). MMS uses BER to encode ASN.1 data before transmission. As we will be decoding BER code, we will explain BER encoding in the next section.

2.6 BER

The Basic Encoding Rules (BER) is one of the original sets of encoding rules specified by the ASN.1 standard for encoding abstract information into a concrete data stream. The rules, collectively referred to as a transfer syntax in ASN.1 parlance, specify the exact octet sequences which are used to encode any given data item before it is transmitted over a network. The BER syntax is defined by the ITU-T’s X.690 standards document, which is part of the ASN.1 document series[3]. In addition to BER there are three alternative encodings, Canonical Encoding Rules (CER), Distinguished Encoding Rules (DER) and Packet Encoding Rules (PER), but as these are not relevant for this report we will not discuss them further.

[3] A description of BER can also be found at http://www.vijaymukhi.com/vmis/ber.htm

Bit number	7	6	5	4	3	2	1	0	Implication
	0	0							UNIVERSAL
	0	1							APPLICATION SPECIFIC
	1	0							CONTEXT SPECIFIC
	1	1							PRIVATE
			0						primitive data type
			1						constructed data type
				X	X	X	X	X	numeric identifier

Table 3: Description of the BER identifier.

The BER identifier is described in Table 3. The seventh and sixth bits are combined to denote the class of the ASN.1 tag. The sixth bit of the identifier indicates whether the represented data type is a primitive or constructed one. The remaining X'ed bits of the identifier represent a class number which is associated with a specific data type.

BER is an self-identifying and self-delimiting encoding scheme, which means that each data value can be identified, extracted and decoded individually. Each data element is encoded using a triplet consisting of a type identifier (tag), a length description and the actual data element1 The use of such a triplet for encoding is commonly referred to as a Tag-Length-Value (TLV) encoding. The specifics of a tag are described in section 2.5. A generic triplet is depicted below:

```
[identifier (tag)] [length (of the contents)] [contents]
```

The use of TLV encoding allows any receiver to decode the ASN.1 information from an incomplete information stream, without any requiring any pre-knowledge of the size, content or semantic meaning of the data, assuming that the communicating parties share the same context specific module definitions.

BER uses the unique code assigned to an ASN.1 data type as an identifier for a data type. This identifier is encoded as one or more bytes of every data type and creates the tag. We can distinguish between two data types using these identifiers. The identifier is well-structured to allow the representation of three levels of information within one such code. All information encoded into an identifier is illustrated in Table 3. On the highest level, represented by the highest-order two bits of the tag octet(s), the class of the data type is encoded. The third highest bit of the identifier indicates whether the represented data type is a primitive or constructed one. A constructed data type can be seen as a complex or compound data type hierarchically based on one or more primitive data types. Data types are described in section 2.5. The remainder of the identifier is a numeric tag associated with a data type within a class. Tags ranging from 0 to 30 can be associated with the remaining 5 bits of the octets. For larger tags, these 5 bits are set to 111111, and one or more subsequent octets are used to encode the tag.

2.7 Transport Layer

The OSI Transport layer protocol (ISO-TP) is described in ISO 8073[15] and depicted in figure 2.6. It manages sequencing, error control and error recovery on an end-to-end basis to ensure complete and correct data transfer. The OSI Transport layer protocol also performs the translation of transport addresses to network addresses and is responsible for flow control, multiplexing and demultiplexing of transport connections. There are five transport layer protocols defined in the OSI suite, enumerated from Transport Protocol Class 0 through Transport Protocol Class 4. These protocols increase in complexity from 0-4. We will give a short description of each class before we look further into Transport Protocol Class 4 which is relevant for this memo. We will use TCP as a basis of comparison as most readers will be more familiar with TCP:

- Transport Protocol Class 0 (TP0) performs segmentation (fragmentation) and reassembly functions. TP0 discerns the size of the smallest maximum PDU supported by any of the underlying networks, and segments the packets accordingly. The packet segments are reassembled at the receiver.
- Transport Protocol Class 1 (TP1) performs segmentation (fragmentation) and reassembly, plus error recovery. TP1 sequences PDUs and will retransmit PDUs or re-initiate the connection if an excessive number of PDUs are unacknowledged.
- Transport Protocol Class 2 (TP2) performs segmentation and reassembly, as well as multiplexing and demultiplexing of data streams over a single virtual circuit.
- Transport Protocol Class 3 (TP3) offers error recovery, segmentation and reassembly, and multiplexing and demultiplexing of data streams over a single virtual circuit. It will only work with connection-oriented communications, in which a session must be established before any data is sent. TP3 also sequences PDUs and retransmits them or re-initiates the connection if an excessive number are unacknowledged.
- Transport Protocol Class 4 (TP4) offers error recovery, performs segmentation and reassembly, and supplies multiplexing and demultiplexing of data streams over a single virtual circuit. TP4 sequences PDUs and retransmits them or re-initiates the connection if an excessive number are unacknowledged. TP4 provides reliable transport service and functions with either connection-oriented or connectionless network service. TP4 is the most commonly used of all the OSI transport protocols, and is similar to the TCP in the TCP/IP suite.

Both TP4 and TCP are designed to provide a reliable connection oriented end-to-end transport service on top of an unreliable network service. This service is sometimes referred to as COTP is other literature. The underlying network service may lose packets, store them, deliver them in the wrong order or even create duplicate packages. Both protocols have to be able to deal with network problems, e.g., a sub network stores valid packets and sends them at a later time. TP4 and TCP have a connection, transfer and a disconnection phase. The principles of these phases are also quite similar. One difference between TP4 and TCP is that TP4 uses ten different TPDU (Transport Protocol Data Unit) types whereas TCP knows only one. This makes TCP a simpler protocol, but adds the cost of increased overhead as every TCP header has to contain all possible header fields. Therefore the TCP header is at least 20 bytes long whereas the TP4 header takes at least 5 bytes. TP4 provides a connection oriented message based service, whereas TCP provides a connection oriented byte based service. Therefore, sequence numbers enumerate and confirm TPDUs on an per message level rather than on a per byte level. Next, sequence numbers are not initiated from a clock counter as in TCP, but rather start from 0. A destination reference number is used to distinguish between connections. This number is similar to the destination port number in TCP, but here the reference number maps onto the port number and can be chosen randomly or sequentially. Another difference is the way both protocols react in case of a call collision. TP4 opens two bidirectional connections between the TSAPs whereas TCP opens just one connection. TSAP is similar to the port concept used in TCP. TP4 uses a different flow control mechanism for its messages, it also provides means for quality of service negotiation and measurement.

3 Analysis of MMS

We will in this chapter look closer at the security of the Manufacturing Message Specification (MMS). We will particularly look into the construction of MMS packages and how they may be altered and forged.

3.1 Analysis of MMS Communication

ISO 8823 states that the ISO OSI transport protocol exchanges information between peers in discrete units of information called transport protocol data units (TPDUs) [1]. This is a fundamental difference between the TCP and the network service expected by Transport Protocol Class 0 (TP 0). The difference, as described in section 2.7, is that TCP manages a continuous stream of octets, with no explicit boundaries, while TP0 expects information to be sent and delivered in discrete objects termed network service data units. Therefore RFC 1006 [16] describes that all TPDUs shall be encapsulated in discrete units called TPKTs. The TPKT layer is used to provide these discrete packets to the OSI Connection Oriented Transport Protocol (COTP) on top of TCP.

We have intercepted some packages using Wireshark. As Wireshark does not support the MMS protocol we were forced to manually decode the MMS PDUs. We are interested to see what kind of information we can identify in these messages. We also wish to look closer at the construction of an MMS message. When looking at MMS communication in Wireshark we found the MMS protocol stack depicted in Figure 4.

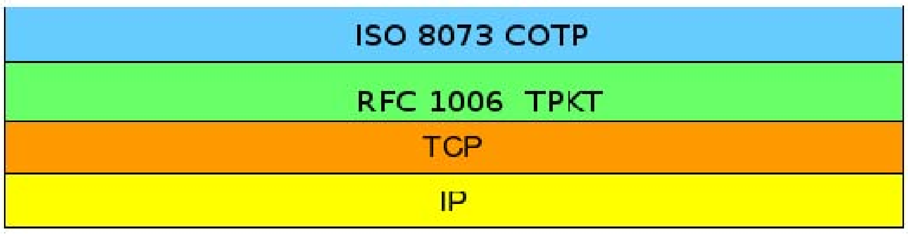


Figure 4: The MMS communication stack as Wireshark detects it

As we see in Figure 4, there are two protocols running on top of TCP. Above TCP we find Transport packet (TPKT), which is a packet format used to emulate the ISO transport services Connection Oriented Transport Protocol (COTP) on top of TCP. RFC 1006 [16] describes how to implement ISOs transport protocol class 0 on top of TCP. ISO 9506 stipulates the use of OSI transport class 4 in conjunction with MMS. Nevertheless RFC 1006 describes the use of OSI transport class 0 to emulate an ISO Transport Service on top of the TCP. The reason for using ISOs OSI transport class 0 on top of TCP/IP instead of transport class four is that transport class 0 achieves identical functionality as transport class 4 when running on top of TCP. The TCP layer provides reliable transport service through error detection and retransmission. It also handles segmentation and reassembly of PDUs. As TCP provides all these properties as part of its service to the next layer there is no reason to implement them again in the “ISO” layer.

A TPKT consists of two parts: a packet-header and a TPDU. The format of the header is constant regardless of the type of packet and is as follows:

```
Field size:  <     8 bits    ><    8 bits    ><              16 bits            >

             +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
Field name:  |      vrsn     |    reserved   |          packet length         |
             +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

The field labelled `vrsn` is the version number which according to RFC 1006 always is three. The next field, reserved, is reserved for further use. The last field is the packet length. This field contains the length of entire packet in octets, including packet-header. The maximum TPDU size is 65531 octets, with a payload of maximum 65524 octets.

According to Wireshark we find COTP above the TPKT layer. RFC 0905 [15] describes the ISO 8073 specification. The COTP data transport PDU is described below.

```
               <          header          ><              body           >

Byte number      1       2           3          4       5            ... end
               +----+-----------+-----------+------------ - - - - - -------+
Field name     | LI |  T    CDT |  TPDU-NR  | User Data                    |
               |    | 1111 0000 |  and EOT  |                              |
               +----+-----------+-----------+------------ - - - - - -------+
```

The header length in octets is indicated by a binary number in the length indicator (LI) field. This field has a maximum value of 254 (1111 1110)[4] . The next field is divided into two parts, first the PDU type specification (T), which describes the structure of the rest of the PDU, e.g., Data Transfer (1111) as described above. The PDU type is encoded as a four bit word. The full list of codes for data types can be found in [15]. The second part is the credit part (CDT) which is used to indicate a reliable transport service, but this is always set to 0000 as TP 0 does not offer reliable transport. The third field contains the TPDU number and an end of transfer indication flag. In all data transfer packets the EOT flag is set and the TPDU number is zero, this might be because the service relies on TCP sequence numbering on the TCP layer, but we have not found any written documentation to support this claim.

We wish to note that there is no reference to ACSE in our packet dump. We verified through Wireshark's documentation that ACSE is a supported protocol [17]. That leaves us with two possible conclusions:

1. The MMS protocol uses an implementation of ACSE which is not in conformance with the standard, which leaves Wireshark unable to decode the packet layer.
2. The implementers of the MMS protocol have omitted the ACSE layer when implementing the protocol.

As we will show in section 3.2, we are fully capable of decoding the whole payload of the COTP PDU to MMS structured ASN.1 text. We therefore conclude that the current implementation of the MMS have omitted the ACSE layer, making the ACSE authentication facilities forfeit. This means that there are no authentication or access control facilities at the lower layers of the MMS stack.

[4] The value 255 (1111 1111) is reserved for possible extensions [32]

3.2 Decoding MMS Communication

Now knowing the underlying protocols which MMS is running on, we will study the MMS message communication between the MMS client and the MMS server and try to determine if there are any signs of security mechanisms. We have used the setup depicted in figure 4.2, where the AC 800 M client software is running on the Windows XP workstation. The AC 800 M runs an MMS server[5] . We used the client software to create a small program which we downloaded to the controller over MMS. The program was a very simple *counting upwards application* as described in C code below:

```
int i=0;
while(TRUE)
{
     i=i+1;
}
```

The controller will now report the value of `i`, back to the client at regular intervals using MMS. Once the program was downloaded to the controller and running, we used Wireshark to capture MMS communication on the network. The first thing we noticed when we examined the packet dump in Wireshark, was that there is a pattern in packet communication repeating it self in a period of eight. This pattern was first identified by packet sizes repeating themselves at a period of eight[6].

[5] For this experiment and further testing the server has IP address 193.75.73.55 and the client 193.75.73.36.

[6] Later we discovered that the packages increased in size by one byte, but as we shall see, this increase is due to larger invokeID numbers over two octets.

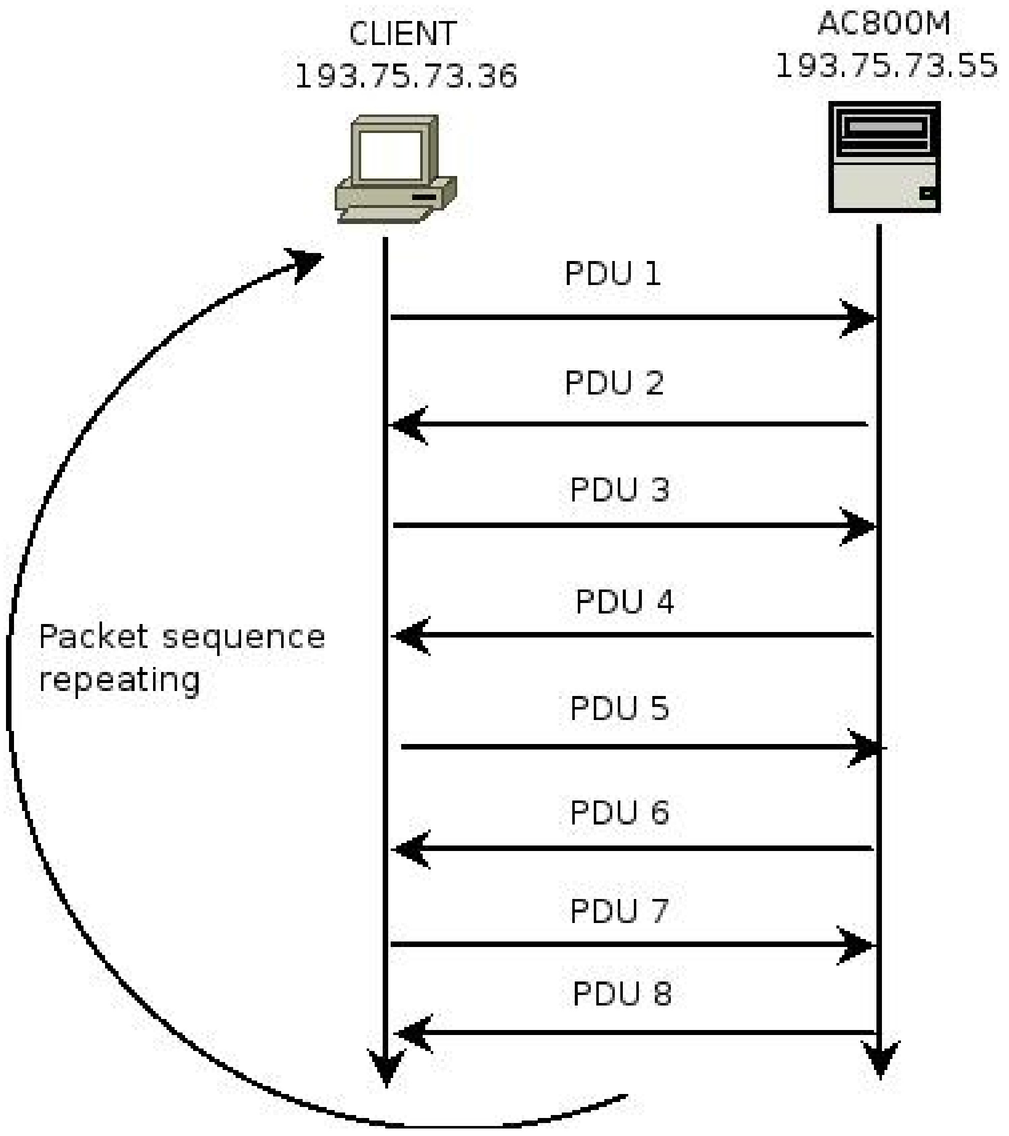
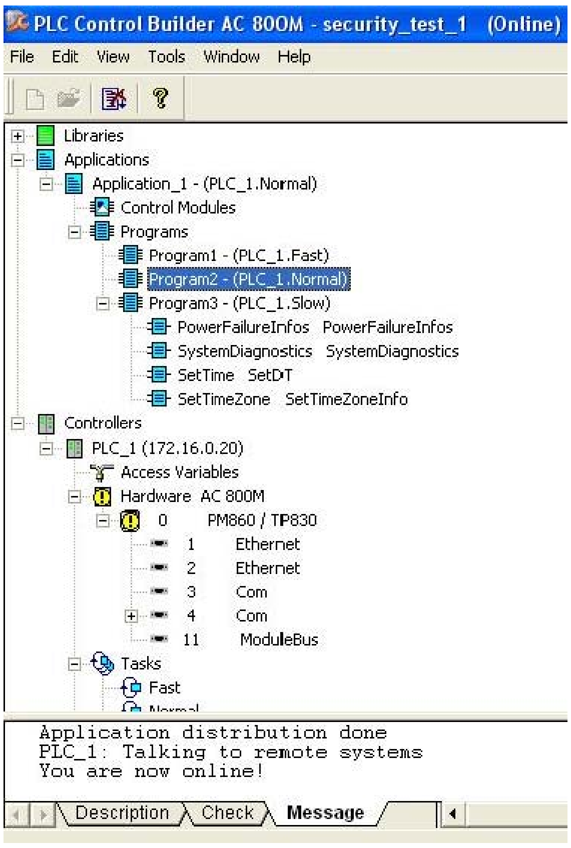


Figure 5: MMS communication between a controller and a client workstation

We wish to note that we chose an arbitrary packet in our packet dump as our starting point and decoded packages sequentially from that point. We chose this strategy to simulate an attacker tapping into a network at an arbitrary point in time. So we have no knowledge of what is the correct staring point in our packet dump. In Figure 6 we depict the program tree as it is represented in ABB’s Control Builder application which runs on the client workstation.


PLC Control Builder AC 800M - security_test_1 (Online)
File Edit View Tools Window Help
Libraries
Applications
Application_1 - (PLC_1.Normal)
Control Modules
Programs
Program1 - (PLC_1.Fast)
Program2 - (PLC_1.Normal)
Program3 - (PLC_1.Slow)
PowerFailureInfos PowerFailureInfos
SystemDiagnostics SystemDiagnostics
SetTime SetDT
SetTimeZone SetTimeZoneInfo
Controllers
PLC_1 (172.16.0.20)
Access Variables
Hardware AC 800M
0 PM860 / TP830
1 Ethernet
2 Ethernet
3 Com
4 Com
11 ModuleBus
Tasks
Fast
Description Check Message

```
Application distribution done
PLC_1: Talking to remote systems
You are now online!
```

Figure 6: Screendump from ABB's Control Builder application

3.2.1 The First PDU

We will now look closer at the first PDU and attempt to decode it. We have extracted the payload of the COTP package at our randomly chosen starting point. We know from ABBs homepages that the AC 800 M uses MMS.

```
a0 41 02 01 7b a4 3c a1 3a a0 38 30 0c a0 0a
80 08 24 4d 53 47 24 31 24 24 30 15 a0 13 80
11 24 48 57 53 34 35 38 35 34 33 32 30 3a 4e
4f 52 4d 30 11 a0 0f 80 0d 24 4d 53 47 24 35
35 32 36 35 38 39 36
```

The package above is encoded in BER's TLV format. We know this from [2] and [3] which are white papers publicly available on the Internet. We must therefore use the decoding rules described in section 2.6 to decode each TLV pair. When decoding this first PDU, with the help of the MMS syntax module [18] we found that this is a confirmed-Request PDU. This confirmed-Request PDU contains an integer id named invokeID with value 123 and confirmedServiceRequest for a read operation. The read-request specifies a listOfVariables with three items. Each item is a vmd-specific object name containing the identifier. We decoded these identifiers to:

```
$MSG$1$$
$HWS45854320:NORM
$MSG$55265896
```

We will now go through the decoding process of the first PDU. As all values are given in hexadecimal numbers, we must first convert them to binary numbers before we can use the BER decoding rules for the tags, but we have omitted this step from the listing below as it is very space consuming and only an intermediate calculation. Using Table 3 we decode the first PDU to the following textual ASN.1 structure:

```
a0             CONTENT SPECIFIC constructed nr 0
41             LENGTH=65

  02           UNIVERSAL primitive nr 2 (INTEGER)
  01           LENGTH=1
    7b              123

  a4           CONTENT SPECIFIC constructed nr 4
  3c           LENGTH=60

    a1              CONTENT SPECIFIC constructed nr 1
    3a              LENGTH=58

      a0       CONTENT SPECIFIC constructed nr 0
      38       LENGTH=56

     30        UNIVERSAL constructed nr 16 (SEQUENCE)
     0c        LENGTH=12

       a0      CONTENT SPECIFIC constructed nr 0
       0a      LENGTH=10

         80         CONTENT SPECIFIC primitive nr 0
         08         LENGTH=8

           24  $
           4d  M
           53  S
           47  G
           24  $
           31  1
           24  $
           24  $

     30        UNIVERSAL constructed nr 16 (SEQUENCE)
     15        LENGTH=21

          a0        CONTENT SPECIFIC constructed nr 0
       0a      LENGTH=19

         80         CONTENT SPECIFIC primitive nr 0
         08         LENGTH=17

           24  $
           48  H
           57  W
           53  S
          34  4
          35  5
          38  8
          35  5
          34  4
          33  3
          32  2
          30  0
          3a  :
          4e  N
          4f  O
          52  R
          4d  M

    30        UNIVERSAL constructed nr 16 (SEQUENCE)
    11        LENGTH=17

      a0      CONTENT SPECIFIC constructed nr 0
      0f      LENGTH=15

        80         CONTENT SPECIFIC primitive nr 0
        0d         LENGTH=13

          24  $
          4d  M
          53  S
          47  G
          24  $
          35  5
          35  5
          32  2
          36  6
          35  5
          38  8
          39  9
          36  6
$
```

We observe that MMS utilizes many CONTENT SPECIFIC tags to identify MMS specific data types. As stated earlier we can use the MMS module definition to decode these tags. This module is publicly available for anyone at the SISCO website or through google. Before we continue our decoding, we feel that we should say some words about the MMS syntax module.

As explained in section 2.5 all ASN.1 modules begin with the same syntax. We see form the module top that the module name is MMS and it has the object identifier stated in the curly brackets below. The module exports the MMSPDU declaration, and imports some object definitions from the ISO 8650 ACSE module.

```
MMS { iso standard 9506 part(2) mms-general-module-version(2) }
DEFINITIONS ::=
BEGIN
EXPORTS MMSpdu;
IMPORTS
     AP-title,
     AP-invocation-identifier,
     AE-qualifier,
     AE-invocation-identifier
FROM ISO-8650-ACSE-1
```

Now we will use some excerpts from the MMS module to decode all content specific tags to textual ASN.1, identifying each of the object definitions used in this PDU. We start with the MMS PDU module definition which is exported to the lower layers. The module defines the different types of PDUs available in MMS and an excerpt is printed below. The full MMS module can be found in appendix A.2.3.

```
MMSpdu ::= CHOICE
{
confirmed-RequestPDU  [0] IMPLICIT Confirmed-RequestPDU,
confirmed-ResponsePDU [1] IMPLICIT Confirmed-ResponsePDU,
confirmed-ErrorPDU    [2] IMPLICIT Confirmed-ErrorPDU,

*continues*
}
```

From the MMS PDU definition, we see that we have a choice between a set of PDUs. We decoded the first tag of the PDU hex dump to be a CONTENT SPECIFIC constructed tag with identifier number zero. We see that this maps to a confirmed-RequestPDU. The confirmed-RequestPDU consists of an implicit sequence and is described in ASN.1 syntax below.

```
Confirmed-RequestPDU ::= IMPLICIT SEQUENCE
{
  invokeID       Unsigned32,
  listOfModifier SEQUENCE OF Modifier OPTIONAL,
  confirmedServiceRequest,
  [79] CS-Request-Detail OPTIONAL
}
```

The confirmed-RequestPDU has an invokeID which is an Unsigned32 number. We decoded the invokeID of our first PDU to have the value 123. As seen in the decoded hex values in the beginning of this section. The invokeID is followed by an listOfModifier which is optional and to our knowledge not used in our implementation of MMS. The next tag must be mapped to a confirmedServiceRequest as this is the next identifier in the sequence. An excerpt from the confirmedServiceRequest module is depicted below, for the full module we refer to appendix A.2.2. The CS-Request-Detail is also optional, and not included in the messages we have intercepted.

```
ConfirmedServiceRequest ::= CHOICE
{
status       [0] IMPLICIT Status-Request,
getNameList  [1] IMPLICIT GetNameList-Request,
identify     [2] IMPLICIT Identify-Request,
rename       [3] IMPLICIT Rename-Request,
read         [4] IMPLICIT Read-Request,

*continues*
}
```

The ConfirmedServiceRequest gives a choice between a large number of requests; we have truncated the list after identifying the correct tag. We decoded the tag to be a CONTENT SPECIFIC constructed tag with identifier number 4, which indicates a Read-Request. The Read-Request is an implicit sequence consisting of a specificationWith-Result identifier, which by default is set to false. Default values are not transmitted as they are known to both sender and receiver. The next identifier in the Read-Request sequence is an explicit variableAccessSpecification.

```
Read-Request ::= IMPLICIT SEQUENCE
{
specificationWithResult     [0] IMPLICIT BOOLEAN DEFAULT FALSE,
variableAccessSpecification [1] VariableAccessSpecification
}

VariableAccessSpecification ::= CHOICE
{
listOfVariable   [0] IMPLICIT SEQUENCE OF SEQUENCE
     {
     variableSpecification     VariableSpecification,
     alternateAccess       [5] IMPLICIT AlternateAccess OPTIONAL
     },
variableListName [1] ObjectName
}
```

The variableAccessSpecification object definition is a choice between a listOfVariable or a variableListName. We have a CONTENT SPECIFIC constructed tag with identifier number zero, which indicates that our PDU is a confirmed request PDU containing a read request for a list of variables. As we can read from the definition above the listOfVariable is an implicit sequence of sequence(s) of type VariableSpecification. Our decoding does not contain a CONTENT SPECIFIC constructed tag nr five so the optional AlternateAccess tag is not used.

```
VariableSpecification    ::=  CHOICE
{
name                          [0]     ObjectName,
address                       [1]     Address,
variableDescription           [2]     IMPLICIT  SEQUENCE
                {
                address Address,
                typeSpecification TypeSpecification,
                },
scatteredAccessDescription [3]     IMPLICIT
                          ScatteredAccessDescription,
invalidated                   [4]     IMPLICIT  NULL
}
```

We see that there are different ways to address an object, just as described in section 2.2. Our decoding translates to a CONTENT SPECIFIC constructed tag nr 0 which is an Objectname.

```
ObjectName ::= CHOICE
{
vmd-specific [0] IMPLICIT Identifier,
domain-specific [1] IMPLICIT SEQUENCE
     {
     domainId Identifier,
     itemId Identifier
     },
aa-specific [2] IMPLICIT Identifier
}

Identifier ::= VisibleString
```

The ObjectName is a vmd-specific name, as our next tag is CONTENT SPECIFIC constructed tag nr 0. This gives us an identifier which is a VisibleString with the value = `$MSG$1$$`. There are two more objects requested in the listOfVariables. These are also mapped to VisibleString in the same manner as the first object. These have identifiers `$HWS45854320:NORM` and `$MSG$55265896`. To summarize, the first PDU, is a Confirmed-RequestPDU with an invokeID of 123 and a confirmedServiceRequest for reading a list of variables addressed by their vdm-specific identifier.

3.2.2 The Second PDU

We will now continue with decoding the response from the MMS server back to the client. The hex-dump of the response can be seen below.

```
a1 1c 02 01 7b a4 17 a1 15 83 01 01 85 01 ff 85 01
03 85 01 02 83 01 00 85 04 02 bb ae 70
```

We expect this PDU to contain the values requested by the client in the previous packet. When decoding the PDU we get the following values.

- A boolean TRUE
- An integer with value -1
- An integer with value 3
- An integer with value 2
- A boolean FALSE
- The hexadecimal value 0x2bbae70 or 45854320 in the decimal number system

How to interpret these values are currently not clear to us, but hopefully this will become clearer after we decode some more PDUs. But we observe that the decimal number is similar to that found in the identifier $HWS45854320:NORM in the previous request. The PDU decodes to the following ASN.1 structure.

```
a1             CONTENT SPECIFIC constructed nr 1
1c             LENGTH=28

  02           UNIVERSAL primitive nr 2 (INTEGER)
  01           LENGTH=1
    7b              123

  a4           CONTENT SPECIFIC constructed nr 4
  17           LENGTH=23

    a1              CONTENT SPECIFIC constructed nr 1
    15              LENGTH=21

      83       CONTENT SPECIFIC primitive nr 3
      01       LENGTH=1
        01     1

      85       CONTENT SPECIFIC primitive nr 5
      01       LENGTH=1
        ff     -1

      85       CONTENT SPECIFIC primitive nr 5
      01       LENGTH=1
        03     3

      85       CONTENT SPECIFIC primitive nr 5
      01       LENGTH=1
        02     2

      83       CONTENT SPECIFIC primitive nr 3
      01       LENGTH=1
        00     0

      85       CONTENT SPECIFIC primitive nr 5
      04       LENGTH=4

     02        0x2
     bb        0xbb
     ae        0xae
     70        0x70
```

We notice that the first tag is a1, which is a CONTENT SPECIFIC constructed nr one. As seen in the MMSpdu module depicted below, this maps to a Confirmed-ResponsePDU. The full MMS PDU module can be found in section A.3. So we can verify the assumption that this PDU is the response of a previous request PDU.

```
MMSpdu ::= CHOICE
{
  confirmed-RequestPDU  [0] IMPLICIT Confirmed-RequestPDU,
  confirmed-ResponsePDU [1] IMPLICIT Confirmed-Response

*continues*
}
```

A Confirmed-ResponsePDU is described below. It consists of an invokeID and a ConfirmedServiceResponce. The invokeID is the same as for the request in the previous PDU, namely 123. This indicates a linking mechanism between a request-response pair through a shared invokeID. The optional CS-Request-Detail field in not in use.

```
Confirmed-ResponsePDU ::= SEQUENCE
{
  invokeID Unsigned32,
  ConfirmedServiceResponse,
  [79] CS-Request-Detail OPTIONAL
}

ConfirmedServiceResponse ::= CHOICE
{
  status      [0] IMPLICIT Status-Response,
  getNameList [1] IMPLICIT GetNameList-Response,
  identify    [2] IMPLICIT Identify-Response,
  rename      [3] IMPLICIT Rename-Response,
  read        [4] IMPLICIT Read-Response,
*continues*
}
```

Then we see from our decoding that the invokeID tag is followed by a CONTENT SPECIFIC constructed nr 4 tag, a read in the ConfirmedServiceResponse, which maps to a Read-Response.

```
Read-Response ::= SEQUENCE
      {
      variableAccessSpecification [0] VariableAccessSpecification OPTIONAL,
      listOfAccessResult          [1] IMPLICIT SEQUENCE OF AccessResult
      }

AccessResult ::= CHOICE
      {
      failure                     [0] IMPLICIT DataAccessError,
      success                         Data
      }

Data ::= CHOICE
      {
      -- context tag 0 is reserved for AccessResult

      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,

      *continues*
      }
```

The Read-Response has a listOfAccessResults which is an implicit sequence of AccessResult. An AccessResult can either be a failure or success. If the request is a success then we get a choice between different primitive data types in the Data module definition depicted above. The full Data module is included in appendix A.2.3. If we get a failure the response will be an integer error code from the DataAccessError module. The DataAccessError module definition in included in appendix A.2.4 From the decoded PDU, we see that we first get a CONTENT SPECIFIC primitive nr three, which maps to a boolean with value 1 or TRUE. Followed by a CONTENT SPECIFIC primitive nr five, which is an integer with value -1 as all integers are written in two's complement [2]. Then an integer with value 3, and then another integer with value 2 followed by another boolean with value 0 or FALSE and finally the hexadecimal value `0x2bbae70` which converts to 45854320 in the decimal number system. As commented earlier is this the same value as we discovered in one of the vdm-specific identifiers in the previous request. So the two messages are linked in some way but neither of them contain data from the process counting upwards.

3.2.3 The Third PDU

The next PDU captured is printed below:

```
a0 2f 02 01 7c a4 2a a1 28 a0 26 30 24 a1 22
82 20 03 ff 17 52 32 36 38 34 32 38 37 36 41
70 70 6c 69 63 61 74 69 6f 6e 5f 31 02 00 03
01 00 03
```

As we will show below this PDU contains a invokeID with value 124 and an unconstrainedAddress of the OCTET STRING type. The unconstrainedAddress has the value 325523R268421876Application 1203103. This PDU decodes to:

```
    a0              CONTENT SPECIFIC constructed nr 0
    2f              LENGTH=47
```

```
   02             UNIVERSAL primitive nr 2 (INTEGER)
   01             LENGTH=1
   7c             124

   a4             CONTENT SPECIFIC constructed nr 4
   2a             LENGTH=42

     a1                CONTENT SPECIFIC constructed nr 1
     28                LENGTH=40

       a0         CONTENT SPECIFIC constructed nr 0
       26         LENGTH=38

 30         UNIVERSAL constructed nr 16 (SEQUENCE)
 24         LENGTH=36

   a1       CONTENT SPECIFIC constructed nr 1
   22       LENGTH=34

     82           CONTENT SPECIFIC primitive nr 2
     20           LENGTH=32

       03   3
       ff   255
       17   23
       52   R
       32   2
       36   6
       38   8
       34   4
       32   2
       31   1
       38   8
       37   7
       36   6
       41   A
       70   p
       70   p
       6c   l
       69   i
       63   c
       61   a
       74   t
       69   i
       6f   o
       6e   n
       5f   _
       31   1
```

```
02  2
00  0
03  3
01  1
00  0
03  3
```

The first a0 tag indicates that the MMSpdu has another confirmed-RequestPDU. The module is printed below.

```
MMSpdu ::= CHOICE
     {
  confirmed-RequestPDU  [0] IMPLICIT Confirmed-RequestPDU,
  confirmed-ResponsePDU [1] IMPLICIT Confirmed-ResponsePDU,

     * continues *
     }
```

We see that this maps to a confirmed-RequestPDU. The confirmed-RequestPDU consists of an implicit sequence and is described in ASN.1 syntax below. We decoded the invokeID of our second confirmed-Request to have the value 124.

```
Confirmed-RequestPDU     ::=  IMPLICIT  SEQUENCE
{
  invokeID     Unsigned32,
  listOfModifier    SEQUENCE  OF   Modifier  OPTIONAL,
  confirmedServiceRequest,
  [79] CS-Request-Detail OPTIONAL
}
```

Then follows another confirmedServiceRequest as thelistOfModifier is optional and not used in this implementation of MMS. The next tag must be mapped to a confirmed-ServiceRequest as this is the next identifier in the sequence. The CS-Request-Detail is also optional, and not included in the messages we have intercepted. The confirmedServiceRequest is printed below and from the decoding above we se that the next tag, a4, maps to a read which is an implicit Read-Request.

```
ConfirmedServiceRequest ::= CHOICE
{
status       [0] IMPLICIT Status-Request,
getNameList  [1] IMPLICIT GetNameList-Request,
identify     [2] IMPLICIT Identify-Request,
rename       [3] IMPLICIT Rename-Request,
read         [4] IMPLICIT Read-Request,

*continues*
}
```

The Read-Request is an implicit sequence consisting of an specificationWithResult identifier, which by default is set to false. Default values are not transmitted as they are known to both sender and receiver [30]. The next identifier in the Read-Request sequence is a explicit variableAccessSpecification which is depicted below.

```
Read-Request ::= IMPLICIT SEQUENCE
{
specificationWithResult     [0] IMPLICIT BOOLEAN DEFAULT FALSE,
variableAccessSpecification [1] VariableAccessSpecification
}

VariableAccessSpecification ::= CHOICE
{ listOfVariable [0] IMPLICIT SEQUENCE OF SEQUENCE
      {
      variableSpecification VariableSpecification,
      alternateAccess [5] IMPLICIT AlternateAccess OPTIONAL
      },

variableListName [1] ObjectName
}
```

The VariableAccessSpecification is a choice between a listOfVariable or a variableList-Name. From our decoding we se that we have a CONTENT SPECIFIC constructed nr 0 tag which leads to a listOfVariable and further to a VariableSpecification.

```
VariableSpecification ::= CHOICE
        {
        name                            [0] ObjectName,
        address                         [1] Address,
        variableDescription             [2] IMPLICIT SEQUENCE
                 {
                 address                     Address,
                 typeSpecification           TypeSpecification,
                 },
        scatteredAccessDescription      [3] IMPLICIT ScatteredAccessDescription,
        invalidated                     [4] IMPLICIT NULL
        }
```

The VariableSpecification has a CONTEXT SPECIFIC constructed nr one tag points towards a Address. The Address definition contains the address types described in section 2.6.1.

```
Address ::= CHOICE
{
numericAddress         [0] IMPLICIT Unsigned32,
symbolicAddress        [1] IMPLICIT VisibleString,
unconstrainedAddress   [2] IMPLICIT OCTET STRING
}
```

The final tag, 82, which is a CONTEXT SPECIFIC primitive nr two tag, indicates that this address is an unconstrainedAddress of the OCTET STRING type. The unconstrainedAddress has the value `325523R268421876Application 1203103`. The addressing form unconstrainedAddress is described in section 2.6.1. So we assume this PDU is a request for the value stored at the `unconstrainedAddress` `325523R268421876Application_1203103` in the AC 800 M controller. We are not sure if this unconstrainedAddress should be translated into ASCII or interpreted as a hexadecimal value, but as we find a string that makes sense inside the address we have chosen to convert it to ASCII. We recognize a part of the string from figure 4.3, as we can see the text string Application 1 is part of the program tree hierarchy downloaded to the AC 800 M controller. This probably means that there is some logic behind how the unconstrainedAddress is constructed.

3.2.4 The Fourth PDU

The fourth PDU is depicted below.

```
a1 0c 02 01 7c a4 07 a1 05 89 03 00 01 0f
```

When decoding this PDU, we find that it has an invokeID with value 124 and therefore we assume that it is a response to the previous request PDU decoded in section 4.4.3. The request contained an unconstrainedAddress with the value `325523R268421876Application_1203103`. As we will show below, we find that this PDU contains an OCTET STRING with values `0 1 15` or `0115`. We decode the PDU to the following ASN.1 structure as depicted below.

```
a1              CONTENT SPECIFIC constructed nr 1
0c              LENGTH=12

  02            UNIVERSAL primitive nr 2 (INTEGER)
  01            LENGTH=1
  7c            124

  a4            CONTENT SPECIFIC constructed nr 4
  07            LENGTH=7

    a1               CONTENT SPECIFIC constructed nr 1
    05               LENGTH=5

      89        CONTENT SPECIFIC primitive nr 9
      03        LENGTH=3

     00         0
     01         1
     0f         15
```

We notice that the first tag is a1, which decodes to a CONTENT SPECIFIC constructed nr one tag. As seen below, this maps to a Confirmed-ResponsePDU.

```
MMSpdu ::= CHOICE
{
confirmed-RequestPDU  [0] IMPLICIT Confirmed-RequestPDU,
confirmed-ResponsePDU [1] IMPLICIT Confirmed-ResponsePDU,

* continues *
}
```

A Confirmed-ResponsePDU is described below. It consists of an invokeID and a ConfirmedServiceResponse. The invokeID, which decodes to 124, links this

ConfirmedServiceResponse to the previous request through their shared invokeID. The optional field in not in use.

```
Confirmed-ResponsePDU ::= SEQUENCE
{
invokeID Unsigned32,
ConfirmedServiceResponse,

[79] CS-Request-Detail OPTIONAL
}

ConfirmedServiceResponse ::= CHOICE
{
status       [0] IMPLICIT Status-Response,
getNameList  [1] IMPLICIT GetNameList-Response,
identify     [2] IMPLICIT Identify-Response,
rename       [3] IMPLICIT Rename-Response,
read         [4] IMPLICIT Read-Response,
*continues*
}
```

The ConfirmedServiceResponse maps through a read to a Read-Response. An excerpt of the ConfirmedServiceResponse module definition is depicted above. The Read-Response definition is included below.

```
Read-Response     ::=   SEQUENCE
{
variableAccessSpecification  [0] VariableAccessSpecification OPTIONAL,
listOfAccessResult           [1] IMPLICIT SEQUENCE OF AccessResult
}

AccessResult ::= CHOICE
{
failure [0] IMPLICIT DataAccessError,
success     Data
}
```

The AccessResult maps through success to Data, which is depicted below. This module gives us a choice of different primitive data types. We decoded the last tag to be a CONTENT SPECIFIC primitive nr 9 tag, which indicates an OCTET STRING with value 115. So the previous request containing the unconstrainedAddress might be the request for the returned value, 115, which could be stored at the memory location indicated in the unconstrainedAddress field.

```
Data ::= CHOICE
{ --context tag 0 is reserved for AccessResult
  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 OCTET STRING,
  visible-string  [10] IMPLICIT VisibleString,
  binary-time     [12] IMPLICIT TimeOfDay,
  bcd             [13] IMPLICIT INTEGER,
  booleanArray    [14] IMPLICIT BIT STRING
}
```

3.2.5 The Fifth PDU

The fifth PDU contains the following hex-dump.

```
a0 2f 02 01 7d a4 2a a1 28 a0 26 30 24 a1 22 82
20 03 ff 17 52 32 36 38 34 32 31 38 37 36 41 70
70 6c 69 63 61 74 69 6f 6e 5f 31 02 00 03 01 00
03
```

We note that this PDU has the same size and starter tag as the third PDU described in section 3.2.3. We decode the PDU as depicted below.

```
a0              CONTENT SPECIFIC constructed nr 0
2f              LENGTH=47

  02            UNIVERSAL primitive nr 2 (INTEGER)
  01            LENGTH=1
  7c            125

  a4            CONTENT SPECIFIC constructed nr 4
  2a            LENGTH=42

    a1               CONTENT SPECIFIC constructed nr 1
    29               LENGTH=40

      a0        CONTENT SPECIFIC constructed nr 0
      26        LENGTH=38

     30         UNIVERSAL constructed nr 16 (SEQUENCE)
     24         LENGTH=36

       a1       CONTENT SPECIFIC constructed nr 1
       22       LENGTH=34

         82          CONTENT SPECIFIC primitive nr 2
         20          LENGTH=32

           03   3
           ff   255
```

```
                    17  23
                    52  R
                    32  2
                    36  6
                    38  8
                    34  4
                    32  2
                    31  1
                    38  8
                    37  7
                    36  6
                    41  A
                    70  p
                    70  p
                    6c  l
                    69  i
                    63  c
                    61  a
                    74  t
                    69  i
                    6f  o
                    6e  n
                    5f  _
                    31  1
                    02  2
                    00  0
                    03  3
                    01  1
                    00  0
                    03  3
```

If we compare the PDU in section 3.2.3 with this PDU, we find that it, apart from the invokeID, is identical to this one. As the decoding procedure is similar to the decoding of the PDU in section 3.2.3 we choose not repeat it. But we wish to point out that the value of the final tag, 82, indicating an unconstrainedAddress of the OCTET STRING type, is the same is the PDU described in section 3.2.3. The unconstrainedAddress has the value 325523R268421876Application_1203103. This indicates that the AC 800 M controller stores the counting variable at a fixed memory location.

3.2.6 The Sixth PDU

The sixth PDU is printed below and is a response to the request described in the previous section. It relates to the PDU described in section 3.2.4 in the same manner as PDU three did to PDU five. From the decoding we se that the invokeID has been incremented to 125 and that the OCTET STRING value is 00 01 18 or concatenated 118.

```
a1 0c 02 01 7d a4 07 a1 05 89 03 00 01 12

a1              CONTENT SPECIFIC constructed nr 1
0c              LENGTH=12

  02            UNIVERSAL primitive nr 2 (INTEGER)
  01            LENGTH=1
  7c            125
  a4            CONTENT SPECIFIC constructed nr 4
  07            LENGTH

    a1               CONTENT SPECIFIC constructed nr 1
    05               LENGTH=5

      89        CONTENT SPECIFIC primitive nr 9
      03        LENGTH=3

     0          00
     1          01
     12         18
```

We will not include the module definitions in this section as they are similar to those of section 3.2.4. If we compare the hex-dump of the fourth PDU with the sixth PDU, we see that the only hexadecimal numbers that are changing are those in position 4 and 13[7].

```
Position:        0  1  2  3  4  5  6  7  8  9 10 11 12 13

Fourth PDU:    a1 0c 02 01 7c a4 07 a1 05 89 03 00 01 0f

Sixth PDU:     a1 0c 02 01 7d a4 07 a1 05 89 03 00 01 12
```

Since we now know the position of the invoke and actual data byte, inside the MMS PDU, we can use this knowledge to alter a PDU in transit from the Control Builder to the controller.

3.2.7 The Seventh and Eighth PDU

Finishing up the cycle depicted in Figure 5, the seventh and eighth PDU is a request and response pair similar to PDU pairs three and four and five and six. Their invokeID value is 126 and the response reports back a value of 121 so we have clearly identified the variable counting upwards which is reported back to the control builder. After this a new period of eight PDUs will follow. PDU one and two contain the same values as earlier, but we are unable to come up with any good explanation for their cause.

3.3 Security in MMS

After analyzing MMS protocol communications we will in this section look into the security mechanisms defined by the MMS standard. The standard does specify means for access control through accessControlList objects. We quote from the ISO standard [1]:

> *The &accessMethod field for a Named Variable object shall specify the mode of access. If the Address is declarable (and obtainable) using MMS services, the &accessMethod field shall have the value public, and the Address attribute shall be defined and available to MMS clients requesting the attributes of the Named Variable object. Otherwise, the value of this field is a local issue. The public access method shall not be available unless vadr is supported.*

[7] The length bytes in positions 1,3,6,8 and 10 will increase as the invokeID becomes larger than 0xff and OCTET STRING becomes larger than `0xff ff ff`

From the quote above we see that each Named Object Variable has an accessMethod field, which specify the mode of access. According to the standard, if the address is declarable and obtainable using MMS service primitives the accessMethod shall have the value public. Access to all objects can be controlled by a special object, the Access Control List, that tells which client can read, delete or modify the object. On a general level MMS specifies that if the accessMethod is public the following field shall appear and if the &accessMethod is anything but public, the following field shall not appear. But there are some exceptions. An MMS server may declare an MMS variable that exists only at the instant of access. Such a variable does not have an address per se, but is still accessible at that instant. We see that the standard provides mechanisms for access control, but to our knowledge there are no other security mechanisms included in the MMS standard. We quote from Security Considerations section in the ISO standard:

> *When implementing MMS in secure or safety critical applications, features of the OSI security architecture may need to be implemented. This International Standard provides simple facilities for authentication (passwords) and access control. Systems requiring a higher degree of security will have to consider features beyond the scope of this International Standard. This International Standard does not provide facilities for non-repudiation.*

As stated above, MMS itself is not designed with information security in mind. This indicates that security should be enforced at some lower layer, but as we have seen through our analysis of the MMS, there is no security enforced at any layer. The ACSE layer could have offered some security features, as described in section 2.4, but as the ACSE layer is omitted from our implementation those features are forfeit.

According to the ISO standard the MMS protocol should have implemented some simple facilities for password authentication and access control. We wanted to study these mechanisms to see what security they really offer, but as they are at best optional and at worst not implemented they provide no security what so ever. We have through our analysis seen that they do not exist.

When analyzing the MMS we have found no protection against replay attacks. This is a major concern, as anyone with access to the network may sniff up a packet and then replay it on the network at an inappropriate moment. We do not regard the invokeID field as such a mechanism as it is easily changed. We will therefore look closer into the implemented authentication and access control of the MMS in this chapter to see what protection it offers.

We wish to make one final observation in this section. Below we have included the module definition for access control lists for MMS. As we can se below all entries apart from the two first identifiers are optional in the MMS. We feel that by making virtually all access control functionality OPTIONAL the standard disregards the security aspect. As we have seen earlier there is no access control implemented in the lower layers as ACSE is omitted for the MMS stack. Neither have we through our analysis discovered any use or reference to an access control list in our decoded PDUs. This leads us to believe that in this implementation of MMS, the ACCESS-CONTROL-LIST was regarded as optional and therefore not implemented.

```
ACCESS-CONTROL-LIST ::= CLASS {
  &name Identifier,
  &accessControl Identifier,
  &readAccessCondition    [0] AccessCondition OPTIONAL,
  &storeAccessCondition   [1] AccessCondition OPTIONAL,
  &writeAccessCondition   [2] AccessCondition OPTIONAL,
  &loadAccessCondition    [3] AccessCondition OPTIONAL,
  &executeAccessCondition [4] AccessCondition OPTIONAL,
  &deleteAccessCondition  [5] AccessCondition OPTIONAL,
  &editAccessCondition    [6] AccessCondition OPTIONAL,

--The following fields are used to record lists of objects placed
--under the control of this ACCESS-CONTROL-LIST object.
--They will be referred to collectively as the Controlled Object Lists

  &AccessControlLists Identifier OPTIONAL,
  &Domains Identifier OPTIONAL,
  &ProgramInvocations Identifier OPTIONAL,
  &UnitControls Identifier OPTIONAL,

  IF (vadr)
    &UnnamedVariables Address OPTIONAL,
  ENDIF
    IF (vnam)
    &NamedVariables ObjectName OPTIONAL,
      IF (vlis)
        &NamedVariableLists ObjectName OPTIONAL,
      ENDIF
    &NamedTypes ObjectName OPTIONAL,
  ENDIF
  &DataExchanges ObjectName OPTIONAL,
  &Semaphores Identifier OPTIONAL,
  &OperatorStations Identifier OPTIONAL,
  &EventConditions ObjectName OPTIONAL,
  &EventActions ObjectName OPTIONAL,
  &EventEnrollments ObjectName OPTIONAL,
  &Journals ObjectName OPTIONAL,
  IF (cspi)
    &EventConditionLists ObjectName OPTIONAL
  ENDIF
}
```

3.4 MMS Plugin for Wireshark

After performing this analysis, we sent a draft version to members of the Wireshark development community. They created a patch for Wireshark based on our findings related to the implemented MMS protocol stack. The MMS plugin has thus been available since 02. May 2007. To run this release you must download version 0.99.5 (or a later version) of the development code and compile it. This patch enables us to dissect the MMS packets and read their content.

Appendix A MMS Details

A.1 Table of MMS Objects with Description

MMS Model Object	Description
Context	The context object represents the attributes that are exchanged so that the MMS behavior is known to both cooperating applications prior to attempting to use other MMS services.
Virtual Manufacturing Device	The VMD itself can be viewed as the object in which all other MMS objects are contained. It has attributes that reflect general capabilities and a general set of methods that are inherited by all other MMS objects.
Named variables	This class of object is, in general, used for ”real-time” data exchange. Its intended use is for data monitoring, non-historic data reporting, and allowing data to be reported in an unsolicited fashion.
Named Variable List	This class of object is used to aggregate, into a list, other variable objects. It differs from the Named Variable object in that each element in the list (when accessed) returns its own success or failure.
Scattered Access	This class of object is, in general, used for ”real-time” data exchange. It differs in capability from the Named and Named Variable List objects through its external behavior. The differentiating behavior is its ability to aggregate other variable objects and have the external appearance of creating a single coherent variable. The intended use of this object is to allow non-MMS variables to be aggregated into complex MMS objects allowing for the migration of legacy protocols.
Named Type	This class of object is used to create a dictionary of well known, and re-usable, data type definitions which allow complex variables to be created. It allows the consistent definition of data representation and the associated range of values to be defined.
Semaphore	This class of object is intended to be used for the resolution of object/resource contention. The characteristics of this object were developed with the UNIX semaphore/token model as the basis.
Event Condition	This class of object is used to allow network applications to be able to determine the state of a condition (Active, Inactive, or Disabled). The specification does not specify the local processing required to transition the state of the object, but how to use the object to trigger network activity based upon the state transitions. The combination of EventCondition, EventAction, and EventEnrollment object use is intended for the construction of dynamic report by exception network scenarios.

Event Action	Instances of this class of object allows for a set of potential network actions to be defined. These actions are then linked to a particular EventCondition transition through the use of an EventEnrollment object. The combination of EventCondition, EventAction, and EventEnrollment object use is intended for the construction of dynamic report by exception network scenarios.
Event Enrollment	Instances of this class allow a network application to define that a particular EventAction object be performed upon a given Event-Condition state transition. Further, it allows the specification of which network application should be notified of the occurrence of the state transition. The combination of EventCondition, EventAction, and EventEnrollment object use is intended for the construction of dynamic report by exception network scenarios.
Journal	This class of object is used for the exchange of historic or archived information. The object allows for data and network (MMS) transactions to be written into a ”log” type of format that allows for the creation of a Sequence of Events (SOE) recording function. Likewise, the ”logged” information can be retrieved so that applications can regenerate information whose time sequencing is important.
Domain	This class of object has been left intentionally vague within the MMS specification. The standard states that domains represent resources. Therefore, the first question is what is the definition of a ”resource”. The intended definition of resource was any aggregation of objects, data, etc... required to perform a single function. It can contain execution instructions, MMS objects, and other information. In general, the concept was borrowed from the process control industries where batch processing needed to be facilitated. This object allows executive code (programs) and configuration settings to be uploaded/downloaded over the network.
Program Invocations	The behavior of this object was borrowed from the concept of a UNIX Execution Thread. It is the object that is used to start and stop remote application processes.
Operator Station	This object facilitates a poor-man’s Telnet exchange of information. The use of this object is restricted to the exchange of simple text messages for human operators.
File	This class of object is to be used to transfer binary file information. It does not provide record access but transfers files in their entirety.

Table 4: The 15 MMS Objects and a description of their intended use

A.2 MMS Modules

In this section we include the full MMS modules discussed previously.

A.2.1 The MMSPDU Module

```
MMSpdu ::= CHOICE
  {
  confirmed-RequestPDU            [0]     IMPLICIT Confirmed-RequestPDU,
  confirmed-ResponsePDU           [1]     IMPLICIT Confirmed-ResponsePDU,
  confirmed-ErrorPDU              [2]     IMPLICIT Confirmed-ErrorPDU,
  unconfirmed-PDU                 [3]     IMPLICIT Unconfirmed-PDU,
  rejectPDU                       [4]     IMPLICIT RejectPDU,
  cancel-RequestPDU               [5]     IMPLICIT Cancel-RequestPDU,
  cancel-ResponsePDU              [6]     IMPLICIT Cancel-ResponsePDU,
  cancel-ErrorPDU                 [7]     IMPLICIT Cancel-ErrorPDU,
  initiate-RequestPDU             [8]     IMPLICIT Initiate-RequestPDU,
  initiate-ResponsePDU            [9]     IMPLICIT Initiate-ResponsePDU,
  initiate-ErrorPDU               [10]    IMPLICIT Initiate-ErrorPDU,
  conclude-RequestPDU             [11]    IMPLICIT Conclude-RequestPDU,
  conclude-ResponsePDU            [12]    IMPLICIT Conclude-ResponsePDU,
  conclude-ErrorPDU               [13]    IMPLICIT Conclude-ErrorPDU
  }
```

A.2.2 The ConfirmedServiceRequest Module

```
ConfirmedServiceRequest  ::= CHOICE
  {
  status                          [0]     IMPLICIT Status-Request,
  getNameList                     [1]     IMPLICIT GetNameList-Request,
  identify                        [2]     IMPLICIT Identify-Request,
  rename                          [3]     IMPLICIT Rename-Request,
  read                            [4]     IMPLICIT Read-Request,
  write                           [5]     IMPLICIT Write-Request,
  getVariableAccessAttributes     [6]     GetVariableAccessAttributes-Request,
  defineNamedVariable             [7]     IMPLICIT DefineNamedVariable-Request,
  defineScatteredAccess           [8]     IMPLICIT DefineScatteredAccess
                                                   -Request,
  getScatteredAccessAttributes    [9]     IMPLICIT GetScatteredAccessAttributes
                                                   -Request,
  deleteVariableAccess            [10]    IMPLICIT DeleteVariableAccess
                                                   -Request,
  defineNamedVariableList         [11]    IMPLICIT DefineNamedVariableList
                                                   -Request,
  getNamedVariableListAttributes  [12]    IMPLICIT GetNamedVariableListAttributes
                                                   -Request,
  deleteNamedVariableList         [13]    IMPLICIT DeleteNamedVariableList
                                                   -Request,
  defineNamedType                 [14]    IMPLICIT DefineNamedType-Request,
  getNamedTypeAttributes          [15]    IMPLICIT GetNamedTypeAttributes
                                                   -Request,
  deleteNamedType                 [16]    IMPLICIT DeleteNamedType-Request,
  input                           [17]    IMPLICIT Input-Request,
  output                          [18]    IMPLICIT Output-Request,
  takeControl                     [19]    IMPLICIT TakeControl-Request,
  relinquishControl               [20]    IMPLICIT RelinquishControl-Request,
  defineSemaphore                 [21]    IMPLICIT DefineSemaphore-Request,
  deleteSemaphore                 [22]    IMPLICIT DeleteSemaphore-Request,
  reportSemaphoreStatus           [23]    IMPLICIT ReportSemaphoreStatus-Request,
  reportPoolSemaphoreStatus       [24]    IMPLICIT ReportPoolSemaphoreStatus
                                                   -Request,
  reportSemaphoreEntryStatus      [25]    IMPLICIT ReportSemaphoreEntryStatus
```

```
                                                  -Request,
initiateDownloadSequence         [26]    IMPLICIT InitiateDownloadSequence
                                                  -Request,
downloadSegment                  [27]    IMPLICIT DownloadSegment-Request,
terminateDownloadSequence        [28]    IMPLICIT TerminateDownloadSequence
                                                  -Request,
initiateUploadSequence           [29]    IMPLICIT InitiateUploadSequence
                                                  -Request,
uploadSegment                    [30]    IMPLICIT UploadSegment-Request,
terminateUploadSequence          [31]    IMPLICIT TerminateUploadSequence
                                                  -Request,
requestDomainDownload            [32]    IMPLICIT RequestDomainDownload-Request,
requestDomainUpload              [33]    IMPLICIT RequestDomainUpload-Request,
loadDomainContent                [34]    IMPLICIT LoadDomainContent-Request,
storeDomainContent               [35]    IMPLICIT StoreDomainContent-Request,
deleteDomain                     [36]    IMPLICIT DeleteDomain-Request,
getDomainAttributes              [37]    IMPLICIT GetDomainAttributes-Request,
createProgramInvocation          [38]    IMPLICIT CreateProgramInvocation
                                                  -Request,
deleteProgramInvocation          [39]    IMPLICIT DeleteProgramInvocation
                                                  -Request,
start                            [40]    IMPLICIT Start-Request,
stop                             [41]    IMPLICIT Stop-Request,
resume                           [42]    IMPLICIT Resume-Request,
reset                            [43]    IMPLICIT Reset-Request,
kill                             [44]    IMPLICIT Kill-Request,
getProgramInvocationAttributes   [45]    IMPLICIT GetProgramInvocationAttributes
                                                  -Request,
obtainFile                       [46]    IMPLICIT ObtainFile-Request,
defineEventCondition             [47]    IMPLICIT DefineEventCondition-Request,
deleteEventCondition             [48]    DeleteEventCondition-Request,
getEventConditionAttributes      [49]    GetEventConditionAttributes-Request,
reportEventConditionStatus       [50]    ReportEventConditionStatus-Request,
alterEventConditionMonitoring    [51]    IMPLICIT AlterEventConditionMonitoring
                                                  -Request,
triggerEvent                     [52]    IMPLICIT TriggerEvent-Request,
defineEventAction                [53]    IMPLICIT DefineEventAction-Request,
deleteEventAction                [54]    DeleteEventAction-Request,
getEventActionAttributes         [55]    GetEventActionAttributes-Request,
reportEventActionStatus          [56]    ReportEventActionStatus-Request,
defineEventEnrollment            [57]    IMPLICIT DefineEventEnrollment-Request,
deleteEventEnrollment            [58]    DeleteEventEnrollment-Request,
alterEventEnrollment             [59]    IMPLICIT AlterEventEnrollment-Request,
reportEventEnrollmentStatus      [60]    ReportEventEnrollmentStatus-Request,
getEventEnrollmentAttributes     [61]    IMPLICIT GetEventEnrollmentAttributes
                                                  -Request,
acknowledgeEventNotification     [62]    IMPLICIT AcknowledgeEventNotification
                                                  -Request,
getAlarmSummary                  [63]    IMPLICIT GetAlarmSummary-Request,
getAlarmEnrollmentSummary        [64]    IMPLICIT GetAlarmEnrollmentSummary
                                                  -Request,
readJournal                      [65]    IMPLICIT ReadJournal-Request,
writeJournal                     [66]    IMPLICIT WriteJournal-Request,
initializeJournal                [67]    IMPLICIT InitializeJournal-Request,
reportJournalStatus              [68]    IMPLICIT ReportJournalStatus-Request,
createJournal                    [69]    IMPLICIT CreateJournal-Request,
deleteJournal                    [70]    IMPLICIT DeleteJournal-Request,
getCapabilityList                [71]    IMPLICIT GetCapabilityList-Request,
fileOpen                         [72]    IMPLICIT FileOpen-Request,
fileRead                         [73]    IMPLICIT FileRead-Request,
fileClose                        [74]    IMPLICIT FileClose-Request,
fileRename                       [75]    IMPLICIT FileRename-Request,
fileDelete                       [76]    IMPLICIT FileDelete-Request,
fileDirectory                    [77]    IMPLICIT FileDirectory-Request,
additionalService                [78]    AdditionalService-Request
}
```

A.2.3 The MMS Data Module

```
Data ::= CHOICE
  {
  -- context tag 0 is reserved for AccessResult

  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 OCTET STRING,
  visible-string                  [10]     IMPLICIT VisibleString,
  binary-time                     [12]     IMPLICIT TimeOfDay,
  bcd                             [13]     IMPLICIT INTEGER,
  booleanArray                    [14]     IMPLICIT BIT STRING
  }
```

A.2.4 The MMS DataAccessError Module

```
DataAccessError ::= INTEGER
  {
  object-invalidated              (0),
  hardware-fault                  (1),
  temporarily-unavailable         (2),
  object-access-denied            (3),
  object-undefined                (4),
  invalid-address                 (5),
  type-unsupported                (6),
  type-inconsistent               (7),
  object-attribute-inconsistent   (8),
  object-access-unsupported       (9),
  object-non-existent             (10)
  }
```

A.3 The MMS initiate-Request/Response PDU

```
MMS
  initiate-ResponsePDU
    localDetailCalling: 8187
    proposedMaxServOutstandingCalling: 3
    proposedMaxServOutstandingCalled: 3
    proposedDataStructureNestingLevel: 127
    mmsInitRequestDetail
      proposedVersionNumber: 1
      Padding: 5
        1... .... = str1: True
        .1.. .... = str2: True
        ..1. .... = vnam: True
        ...0 .... = valt: False
        .... 1... = vadr: True
        .... .0.. = vsca: False
        .... ..0. = tpy: False
        .... ...0 = vlid: False
        0... .... = real: False
        ..0. .... = cei: False
      Padding: 3
      servicesSupportedCalling: EC00183f0ff41003010090
        1... .... = status: True
        .1.. .... = getNameList: True
        ..1. .... = identify: True
        ...0 .... = rename: False
        .... 1... = read: True
        .... .1.. = write: True
        .... ..0. = getVariableAccessAttributes: False
        .... ...0 = defineNamedVariable: False
        0... .... = defineScatteredAccess: False
        .0.. .... = getScatteredAccessAttributes: False
        ..0. .... = deleteVariableAccess: False
        ...0 .... = defineNamedVariableList: False
        .... 0... = getNamedVariableListAttributes: False
        .... .0.. = deleteNamedVariableList: False
        .... ..0. = defineNamedType: False
        .... ...0 = getNamedTypeAttributes:False
        0... .... = deleteNamedType: False
        .0.. .... = input: False
        ..0. .... = output: False
        ...1 .... = takeControl: True
        .... 1... = relinquishControl: True
        .... .0.. = defineSemaphore: False
        .... ..0. = deleteSemaphore: False
        .... ...0 = reportSemaphoreStatus: False
        0... .... = reportPoolSemaphoreStatus: False
        .0.. .... = reportSemaphoreEntryStatus: False
        ..1. .... = initiateDownloadSequence: True
        ...1 .... = downloadSegment: True
        .... 1... = terminateDownloadSequence: True
        .... .1.. = initiateUploadSequence: True
        .... ..1. = uploadSegment: True
        .... ...1 = terminateUploadSequence: True
        0... .... = requestDomainDownload: False
        .0.. .... = requestDomainUpload: False
        ..0. .... = loadDomainContent: False
        ...0 .... = storeDomainContent: False
        .... 1... = deleteDomain: True
        .... .1.. = getDomainAttributes: True
        .... ..1. = createProgramInvocation: True
        .... ...1 = deleteProgramInvocation: True
        1... .... = start: True
        .1.. .... = stop: True
```

```
..1. .... = resume: True
...1 .... = reset: True
.... 0... = kill: False
.... .1.. = getDomainAttributes: True
.... ..1. = obtainFile: True
.... ...0 = defineEventCondition: False
0... .... = deleteEventCondition: False
.0.. .... = getEventConditionAttributes: False
..0. .... = reportEventConditionStatus: False
...1 .... = alterEventConditionMonitoring: True
.... 0... = triggerEvent: False
.... .0.. = defineEventAction: False
.... ..0. = deleteEventAction: False
.... ...0 = getEventActionAttributes: False
0... .... = reportActionStatus: False
.0.. .... = defineEventEnrollment: False
..0. .... = deleteEventEnrollment: False
...0 .... = alterEventEnrollment: False
.... 0... = reportEventEnrollmentStatus: False
.... .0.. = getEventEnrollmentAttributes: False
.... ..1. = acknowledgeEventNotification: True
.... ...1 = getAlarmSummary: True
0... .... = getAlarmEnrollmentSummary: False
.0.. .... = readJournal: False
..0. .... = writeJournal: False
...0 .... = initializeJournal: False
.... 0... = reportJournalStatus: False
.... .0.. = createJournal: False
.... ..0. = deleteJournal: False
.... ...1 = getCapabilityList: True
1... .... = fileOpen: True
.1.. .... = fileRead: True
..1. .... = fileClose: True
...1 .... = fileRename: True
.... 1... = fileDelete: True
.... .0.. = fileDirectory: False
.... ..0. = unsolicitedStatus: False
.... ...0 = informationReport: False
1... .... = eventNotification: False
.0.. .... = attachToEventCondition: False
..0. .... = attachToSemaphore: False
...1 .... = conclude: False
.... 0... = cancel: False
```

References

[1] "Industrial automation systems, Manufacturing Message Specification. Part 1," ISO 9506-1:2003(E) ISO 2003.

[2] System Integration Specialists Company (SISCO), "Overview and Introduction to the Manufacturing Message Specification (MMS)," 6605 19,5 Mile Road, Sterling Heights, MI 48314-1408, USA 1995.

[3] H. Falk and J. Robbins, "An Explanation of the Architecture of the MMS standard," SISCO, 6605 19,5 Mile Road, Sterling Heights, MI 48314-1408, USA 1995.

[4] H. Falk and D. M. Burns, "MMS and ASN.1 Encoding," SISCO, 6605 19,5 Mile Road, Sterling Heights, MI 48314-1408, USA 2001.

[5] L. Floyd and D. Ronald, "MANUFACTURING AUTOMATION PROTOCOL.," *Conference Record - International Conference on Communications*, pp. 620 - 624, 1985.

[6] "Information processing systems - Open Systems Interconnection, Service Definition for Association Control Service Element," ISO 8649-8649:1996 1996.

[7] "Information processing systems - Open Systems Interconnection, Protocol Specification for Association Control Service Element.," ISO 8650-1:1996 1996.

[8] "X.217 : Information technology - Open Systems Interconnection - Service definition for the Association Control Service Element.," X. 217 (1995 E) ITU-T Recommmendation 1995.

[9] "X.226 : Information technology - Open Systems Interconnection - Connection oriented pressentation protocol: Protocol specification.," X.226 (1994 E) ITU-T Recommmendation 1994.

[10] "Information technology - Open Systems Interconnection - Connection-oriented presentation protocol: Protocol specification.," ISO/IEC 8823-1:1994 ISO 1994.

[11] M. T. Rose, *The open book: a practical perspective on OSI*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1990.

[12] "X.680 :Information technology - Abstract Syntax Notation One (ASN.1): Specification of basic notation.," X.680 (2002 E) ITU-T 2002.

[13] M. Subramanian, *Network Management: An Introduction to Principles and Practice*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1999.

[14] J. Larmouth, *ASN.1 complete*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2000.

[15] A. M. McKenzie, "RFC 905: ISO Transport Protocol specification ISO DP 8073," 1984.

[16] M. T. Rose and D. E. Cass, "RFC 1006: ISO transport services on top of the TCP: Version 3," 1987.

[17] "Wireshark Wiki on ACSE," Wireshark 2006, http://wiki.wireshark.org/ACSE.

[18] "SISCO MMS Syntax," System Integration Specialists Company (SISCO) 1994, http://www.sisconet.com/downloads/mms_abstract_syntax.txt.