Lightweight Message Construction Protocol (LMCP) Implementation Guide

Matt Duquette

13 July, 2012

AVTAS Laboratory

Air Vehicles Directorate

Air Force Research Laboratory

Wright-Patterson AFB, Ohio

1. Introduction

The Lightweight Message Construction Protocol (LMCP) is a standard that serves two purposes. First, it defines a structure for common structured data and a process for serializing objects based on those types. Secondly, it defines a method for encapsulating objects for transmission between applications. This specification describes the structure of data, without specifying how applications implement the handling instantiations of those data types.

Applications that implement this specification can send and receive objects regardless of the operating system, platform, or programming language used. LMCP is a simple and extensible specification, so it can be implemented without a central runtime, proprietary libraries, or with the complexity of other protocols such as HLA and DIS.

This system offers several advantages to the end-user, including

- A design independent of language, platform, and transport protocol.
- Object-oriented approach through structured data types, including support for null objects.
- A high level of customizability through user-defined data models.
- Support for variable length arrays and nested objects.
- Simple and open design, free of proprietary code and requiring no runtime infrastructure or special libraries.

LMCP also defines the structure of a message. A message is a LMCP object which is encapsulated with header and footer items to enable communication of LMCP objects between applications.

LMCP allows developers to create custom data types (structs) easily through a message data model (MDM). Custom classes can be created for a given MDM through automatic code generation or other means to recognize and handle messages.

2. Data Model

LMCP implements an object-oriented data model. The data model describes all of the LMCP data types that are valid for a given LMCP network. An LMCP struct is a structure of other valid LMCP structs, similar to how a class is defined in an object-oriented computer language. Only objects that are reserved by this standard or declared in the same Message Data Model (MDM) may be members of a struct. Section 5 describes how a data model is created.

2.1 Primitive Types

LMCP defines several primitive data types that can be used by LMCP objects, as shown in Table 1. These types are the basis for forming meaningful data structures that are described by the MDM.

2.2 Arrays

Data types defined in an MDM may contain array data in the form of fixed or variable length arrays. An array data type is a list of data of the same type. Arrays can contain only one type of data. Arrays may contain null values if the type is not a primitive data type (i.e. a structured data type).

LMCP reserves a special array data type, “string”. A string is a variable length array of char (see Table 1).

2.3 Structured Data Types (Structs)

The MDM has a listing of the structured data types. A structured data type is analogous to an object-oriented class. Structured types can contain primitives, arrays, or other structured data types. LMCP supports inheritance of structs. See Section 5 for information on how to define structs.

Table 1. LMCP Primitive Data Types

Type	Size (bytes)	Definition and Limits
bool	1	True (1) or false (0)
byte	1	0 to 255
char	1	values according to ASCII table (http://en.wikipedia.org/wiki/ASCII)
real64	8	2.2e-308 to 1.8e308
real32	4	1.2e-38 to 3.4e38
int64	8	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
int32	4	-2,147,483,648 to 2,147,483,647
int16	2	-32,768 to 32,767
uint32	4	0 to 4,294,967,295
uint16	2	0 to 65,535
string	variable	a variable length array of char types

A special struct, *LmcpObject*, is reserved by LMCP. LmcpObject denotes that any valid LMCP struct can be used in the corresponding field. This is analogous to the "Object" base type used in many object-oriented computer languages.

2.4 Enumerations (Enums)

Enumerations are named representations of data that can be used in struct fields. The MDM lists a set of enums that can be used in structs. Enums can contain any valid int32 value. Enum entries can be explicitly assigned a value. If no value is specified in the MDM, the enum entry is assigned a value according to the position in the enum entry list, starting at zero.

3. Object Serialization

An LMCP object (an instance of an LMCP structured data type) is serialized by placing byte representation of all object contents into a buffer. A serialized LMCP object shall be readable by any other application that implements this standard. De-serialization of object bytes shall yield an object of identical content of the source object. Serialization occurs in declaration order; all members of an LMCP object are converted to bytes and placed in a byte array in the order that they are declared by the MDM.

Data is serialized according to the rules below.

- If a data member is a primitive type (those listed in Table 1, with the exception of *string*), its bytes are serialized using the big-endian (most significant byte first) byte order.
- If a data member is an enum, it is serialized according to the rules for int32.
- A struct data type is preceded by a 1-byte boolean denoting its existence. If the object is null, the byte is set to zero, otherwise it is set to 1. If the object is non-null, the 8-byte series name, a 4-byte unsigned integer (uint32) denoting the LMCP object type, and a 2-byte unsigned integer (uint16) denoting the series version number follows. Then the members of the struct are serialized using the rules of this section.

- A variable-length array is preceded by a (uint16) value denoting the length of the array. This value is the number of elements (not the number of bytes) in the array. The data contained in the array is then serialized according to the preceding rules. Arrays may be of zero length, but may not be null. If an array is zero length, the preceding length value shall be zero. Arrays may contain null elements if it contains objects of a non-primitive type. Strings are considered as variable length arrays of char.

4. Messages

A *message* is an encapsulated and serialized LMCP object. Any struct can be packaged as a message. A message contains a header, message body, and checksum. In this section, the data object that forms the body of the message is referred to as the *root object*.

4.1 Header

The header contains information that is used by applications to identify and process messages. The header is 8 bytes long and contains the following data:

- **Control Sequence (char[4]).** The header starts with a 4-byte char sequence of "LMCP" (0x4c4d4350) that is used to indicate the start of a new LMCP message.
- **Length (uint32).** Indicates the length, in bytes, of the root object.

4.2 Root Object

The root object is a single LMCP struct. When serializing the root-level object, the serialization rules given in Section 3 are used.

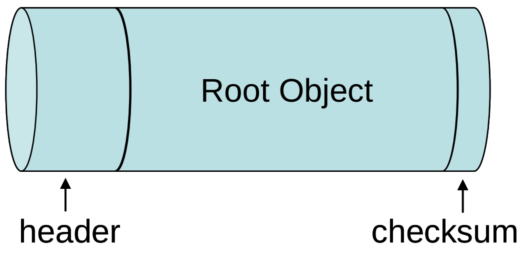

Figure 1. Message Structure

4.3 Checksum

The checksum is an unsigned integer (uint32) that is derived by binary addition of all members of the message except the checksum. The result is truncated to a 32-bit unsigned integer. Applications can choose to not calculate the checksum. In this case, the field shall be set to zero. LMCP-aware applications should recognize the zero value as a non-calculated checksum.

5. Message Data Model

The message data model (MDM) is a list of data structures defined for a given application that all LMCP-aware applications shall implement. It is specified in an XML file as shown in the appendices. The LMCP standard does not specify how the MDM translates into code. However, an XML DTD defines the contents of an MDM.

5.1 Series Name
The SeriesName is used for populating the SeriesName header item when creating a message packet. The SeriesName may be any set of ASCII values of less than 8 chars. Applications implementing LMCP may use the SeriesName value in a message packet as a first-check for MDM compatibility as well as byte-stream integrity.

5.2 Version
The version number is an unsigned 16-bit integer according to the value specified in the MDM. This is an optional setting in the MDM. A value of zero indicates that no versioning is specified.

5.3 Struct Definition
Structured data types are defined within the *StructList* of the MDM. A struct name may be of any combination of letters, numbers, and the underscore ('_') character. A name may not begin with a number. A struct may extend another struct type through the "extends" attribute. For serialization, all members of the antecedent type precede the members of the current type. There is no limit on the number of levels of inheritance. Recursive inheritance is not allowed.

5.4 Field Definition
The members of each struct are described by the MDM. Each field has a name and type. Every instance of a struct contains the members specified by the MDM. Fields can be of any primitive type, array, or another struct specified by the MDM. A struct may also have no fields.

6. Implementation Standards
LMCP defines message content, but leaves the implementation of communication protocols to the user. However, a few suggestions are presented which may help application developers.

6.1 Mixing MDMs
A session should support data from multiple MDMs. Since the MDM series name and series version are included for every struct that is contained in a message, applications should be aware of the presence of data from multiple MDM sources. Not all applications will be able to meaningfully handle all MDMs, but should factory methods should return gracefully when data from unknown MDMs is encountered in a session.

6.2 Streaming Communications
When performing streaming operations, such as TCP/IP, it is best to check the SeriesName in the header each time a read is performed to test for corruption of the data stream.

6.3 Packet-Based Communications
When using packet-based protocols, such as UDP/IP, the use of confirmation messages helps mitigate the possibility of lost data. A manual system for resending "lost" packets must be maintained by the implementing software. If the protocol has a packet length limitation, consider splitting a message into multiple parts.

6.4 Using the Checksum
Implementations of LMCP may choose to ignore the checksum. In the case of TCP, using the checksum may be of little use. Anecdotally, UDP has shown to be reliable under most network conditions, so the checksum may not be used. If the checksum is not to be used, the field shall be zero-filled.

6.5 Using the MDM

The preferred method of utilizing an MDM is to pre-compile libraries or auto-generate code to be compiled with the application. The field of each message should have corresponding “get” and “set” functions to allow access to message data members. Application developers may choose to use the MDM definitions in other ways.

7. XML Representation

7.1 Introduction

LMCP messages can be specified in a data file as binary objects. However, it may be useful to users to view and/or edit LMCP data. XML is a natural language for the storage of hierarchical information and is used by LMCP to define MDM data.

This section describes one way to store LMCP data in an XML format. The rules described in this section can be used to store instantiations of LMCP structs.

An example of an LMCP XML object list is shown in Appendix E.

7.2 Object Representation

Any struct from an MDM may be represented using XML. An object is declared by creating an element with the name of the corresponding LMCP struct. All members of that object are assumed to contain default values as specified in the MDM unless they are listed as children in the XML representation. The series name for the object should be included in the struct’s element as an attribute named “series”.

```
<struct_type series=””>stuff</struct_type>
```

Members of objects are set by listing child elements of the object with a name corresponding to the field in the MDM declaration. Use the following rules for object members:

- Each member fields is listed as its own element under the struct’s element using the field name as the element name.
- If the member is a primitive type, and is not an array, the value is listed as text under an element with the name of the field.

  ```
  <field_name>stuff</field_name>
  ```

- If the member is an array of primitive types , each list item is expressed as a list of child elements of a type that is compatible with the array.

  ```
  <field_name><int>12</int><int>32</int><int>19</int></ field_name>
  ```

- If the member is a struct type, it is listed under the field name using the rules of this section.

  ```
  <field_name><struct_type series=””>stuff</struct_type></field_name>
  ```

- If the member is a list of structs types, each object in the list is listed as a child element of the member element, using the rules of this section to list each object.
- Enumerations are listed using their named value.

Appendix A
Message Data Model

```
<!DOCTYPE MDM SYSTEM 'MDM.DTD'>

<MDM>
    <SeriesName>TestSeries</SeriesName>
    <Namespace>Example</Namespace>
    <Version>1</Version>
    <Comment> MDM Comment </Comment>
    <EnumList>
        <Enum Name=”VehicleCondition”>
            <Entry Name=”GOOD” Value=”27”/>
            <Entry Name=”BAD”  Value=”12”/>
    </EnumList>
    <StructList>
        <!-- put a struct comment here -->
        <Struct Name="Position" Extends=” AnotherMDM/Point”>
            <!-- put a field comment here -->
            <Field Name="Latitude" Type="real64" Default="0" Units=”Degree”>
                  <Comment> or put a field comment here </Comment>
            </Field>
            <Field Name="Longitude" Type="real64" Default="0" Units=”Degree”/>
            <Field Name="Zone" Type="char[10]" Default="a"/>
        </Struct>
        <Struct Name="Status">
            <Field Name="VehicleID" Type="int32" Default="0"/>
            <Field Name="Condition" Type="VehicleCondition" />
            <Field Name="Location" Type=" AnotherMDM/Point" />
            <Field Name="Extra" Type="LmcpObject"/>
        </Struct>
    </StructList>
</MDM>
```

The XML definition of an MDM is straight forward and is shown above as an XML snippet. The following XML entities and attributes are handled in the LMCP specification.

<SeriesName> *required* This is a char[10] value that specifies the name of the MDM series. This can be a maximum of 10 characters. Names less than 10 char long should be terminated with ‘\0’.
<Namespace> *required* This value is used by auto-coding software to establish namespaces or package names under which the message code will exist.
<Version>

optional

Denotes the version of this MDM.

`<EnumList>`

optional

Defines all of the enumerations for this MDM

`<Enum Name="">`

multiple allowed.

This defines an enumeration.

> `Name` *(required)* Name of the enumeration.

`<Entry Name="" Value="">`

multiple allowed, at least one required.

This defines a Enum entry.

> `Name` *(required)* Name of the value
>
> `Value` *(optional)* The value for this entry. If there is no value specified, then the value is set to the index of the entry in the enum.

`<StructList>`

required

Contains all of the messages specified in this MDM and their members.

`<Struct Name="">`

multiple allowed, at least one required.

This defines a struct.

> `Name` *(required)* Name of the struct.
>
> `Extends` *(optional)* Name of the struct that this struct is based on. All members of the antecedent are declared in the struct by using this attribute. To denote an extension of a type in another MDM, use the syntax `<series name>/<struct name>`.
>
> `Series` *(deprecated, used if this struct extends a struct from another MDM)* Series name of the MDM in which the parent struct (denoted by the "Extends" attribute) is defined. This attribute is still supported in LMCP, but the recommended practice is to use the "/" separator in the "Extends" attribute.

`<Field Name="" Type="" Default=""/>`

optional. Used only if the struct contains fields. (Most structs do)

The definition of each field in the message. The order of fields in the XML tree must match the order of the fields in the message type definition.

> `Name` *(required)* is the name of the field.
>
> `Type` *(required)* must be one of the "Type" values listed in **Table 1**, or a type defined by this MDM. If the type is a fixed-length array, then declare the type attribute as <type_name>[xxx], where xxx is the length of the array. If it is a variable length array, declare it as <type_name>[]. If the type is a struct from another MDM, use the syntax `<series name>/<struct name>`.
>
> `Series` *(deprecated, used if the type is from another MDM)* Series name of the MDM in which the struct type is defined. This is deprecated, but still supported in LMCP. The recommended practice is to

use the “/” separator in the type field instead.

`Default` *(optional)* specifies the default value for that field and must be of a valid type for that field, such as an integer, floating point value, or char. If a struct is to be null by default, use **`Default="null".`** Variable length arrays shall always be zero-length by default, so the Default attribute has no effect.

`Units` *(optional)* specifies the units that are used for this field. Typically units (e.g. feet, degrees, etc) apply to real values.

`<!-- comments -->`

Optional.

XML comments are recognized in MDM processing. If an XML comment is in the document, it is assumed that the comment applies to the element directly following the comment. Comments are recognized for the MDM node, an Enum node, an Enum Entry, a Struct node, or a Field node.

To assist in creating easily navigable documentation, the following special tag is declared. Automatic code generators use this to resolve hyperlinks to structs and enums elsewhere in the documentation. To link to a struct or enum in another series, use the syntax [series name]/[struct or enum name] for the “type_name” listed below.

{@link type_name}	Used to create a link to the data type specified by type_name

Appendix B
Message Data Model DTD

```
<?xml version='1.0' encoding='UTF-8'?>

<!--- The MDM Node should contain only the elements that follow -->
<!ELEMENT MDM (SeriesName+, Namespace+, Version?, Comment?, EnumList?, StructList?)>

<!--- This is a text value of less than 10 chars (ascii types) -->
<!ELEMENT SeriesName (#PCDATA) >

<!--- For autocoding purposes.  This is used by languages to set namespaces or directory paths --
>
<!ELEMENT Namespace (#PCDATA)>

<!--- Optional element to specify a version number for the MDM -->
<!ELEMENT Version (#PCDATA)>

<!--- The starting number for messages (must be greater than 10) -->
<!ELEMENT StartID (#PCDATA)>

<!--Comments for the MDM, Struct, or Field -->
<!ELEMENT Comment (#PCDATA)>

<!-- List of enerations that can be used in structs -->
<!ELEMENT EnumList (Enum)*>

<!--- Field definition (used in structs) -->
<!ELEMENT Field (Comment)?>
<!ATTLIST Field
    Default CDATA ''
    Type CDATA #REQUIRED
    Name CDATA #REQUIRED
    Series CDATA ''
    Units CDATA 'None'
  >

<!--- The list of structs -->
<!ELEMENT StructList (Struct)*>

<!--- Struct definition -->
<!ELEMENT Struct (Comment?, Field*)>
<!ATTLIST Struct
    Extends CDATA ''
    Name CDATA #REQUIRED
    Series CDATA ''
  >

<!--- Enumeration definition -->
<!ELEMENT Enum (Comment?, Entry*)>
<!ATTLIST Enum
    Name CDATA #REQUIRED
  >

<!--- Struct entry definition -->
<!ELEMENT Entry (Comment?)>
<!ATTLIST Entry
    Name CDATA ''
    Value CDATA ''
  >
```