DATA MARKUP LANGUAGE

Contact: Wiley Black (WBlack@optics.arizona.edu)

Specification Version: V3.1 Draft

CONTENTS

OVERVIEW

The Data Markup Language (DML) is an efficient and compact version of the Extensible Markup Language (XML), often referred to as a Binary XML. DML specifies a container format that permits the structural components of data storage and interchange to be performed at a generic level while data interpretation is performed at a higher-level with aid from the markup.

XML provides an encoding format and structure that are easily human readable. DML provides an encoding format that is easily computer readable and extensible. See "Data Markup Language – Relationship to Other Encodings" for more discussion of the relationships between DML and XML, as well as a comparison of DML to other binary XMLs, and discussion of conversion between DML and XML.

DML relies on a lookup table called the translation. The translation follows a define-once, use-many model where the translation is defined for a software, file format, storage or communications protocol and many content documents can make use of that same translation. DML Translation information must either be hardcoded or read from the header and separate translation document(s) in order to successfully parse the DML. The DML Translation document is described in detail in the "Data Markup Language – Translation Document" specification. DML is close to what is referred to as a "schema aware" binary XML, but translations provide this functionality instead of schemas. A translation is the minimum schema information necessary to accomplish efficient and compact transmission/storage of data. Translations can also be generated automatically from a complete schema.

DML at its lowest-level supports variable-length formats that require terminators. This is advantageous for streaming data where the length is not known in advance. By building upon DML's structure, software can also utilize the DML structure for fixed or specified data lengths, facilitating indexing or rapid navigation through a large file.

A common unified structure is provided by DML that permits a rich set of primitives to describe data while supporting opaque data that can be handled through extensible software. This is an ideal configuration for a layered-software approach that cleanly supports both simple and arbitrarily complex datasets.

A key feature of DML is that the encoding itself is also extensible. Primitives represent a specific type and encoding of data. DML provides a capability for including additional primitives. The base and extended primitives and their encodings are described in detail in the "Data Markup Language – Primitives" specification. The use of extended primitives requires mention in the DML Translation or DML Header and is discussed in the primitives and translation specifications. The use of custom primitives requires custom software and integration is dependent on the DML parser software, but DML provides all the structure necessary to locate, activate, and configure such custom codecs.

The standardized definition of the DML Translation permits a new kind of DML "generic" reader that does not need a priori information to be able to interpret and display the data markup from the DML content. Thorough standardization of data also permits another new concept with DML – the potential intermixing of data from what would have previously been separate file formats into a single stream without interference, and without encumbering the original file format.

NODES

The lowest-level conceptual unit in DML is a node. The conceptual units in DML can be further divided as:

- Nodes
  - Elements
    - Containers
    - Primitives
  - Attributes
    - Primitives
  - Structural
    - End-container marker
    - End-attributes marker
    - Padding/Comments

The base set of primitives and their encoding is described in the "Data Markup Language – Primitives" specification but includes:

- Unsigned Integer
- String (UTF-8)
- Byte Arrays

DML also provides a mechanism for using more primitives, and a variety are defined. The mechanism allows a DML reader to check whether it recognizes the additional primitives before continuing, allowing DML readers to support only the primitives necessary to accomplish their task. The primitives mechanism allows DML to select a particular kind of encoding/decoding (codec) for the additional primitives.

DML structure and the base primitives are close to endian-neutral, although there is an element of big-endian in some instances. Extended primitives are available in either endianness.

ENCODING

The fundamental unit in DML is the node, and everything in the DML stream constitutes a node or a part of a node. Most of the construction of DML structure comes from the "Compact Unsigned Integer" encoding described next. From the compact unsigned integer, we can describe node encoding.

COMPACT UNSIGNED INTEGERS

The compact unsigned integer (Compact-32) is a variable-length encoding similar to the UTF-8 encoding system used for strings. DML is designed so that the most frequently occurring values are small numbers, i.e. between 0 and 127. The Compact-32 provides a single-byte encoding for those values, but allows extension up to 32-bits in less frequent cases needed for extensibility. When read, the value is zero padded back into a 32-bit unsigned integer. When multiple bytes are used, a big-endian sequence is employed. The bits of the integer are represented as "x" entries in the encoding table:

Binary Encoding (Compact-32)	Minimum Value	Maximum Value
1xxx xxxx	0x00	0x7F
01xx xxxx xxxx xxxx	0x80	0x3FFF
001x xxxx xxxx xxxx xxxx xxxx	0x4000	0x1FFFFF
0001 xxxx xxxx xxxx xxxx xxxx xxxx xxxx	0x200000	0x0FFFFFFF
0000 1000 xxxx xxxx xxxx xxxx xxxx xxxx xxxx xxxx	0x10000000	0xFFFFFFFF

We refer to the above encoding as a Compact-32 value. The decoder examines the first byte retrieved in order to determine how many additional bytes must be read to reconstruct the 32-bit value. Through careful design, the most frequent answer is "no additional bytes", achieving compactness. Data Markup Language also makes use of a Compact-64 value that follows the same pattern:

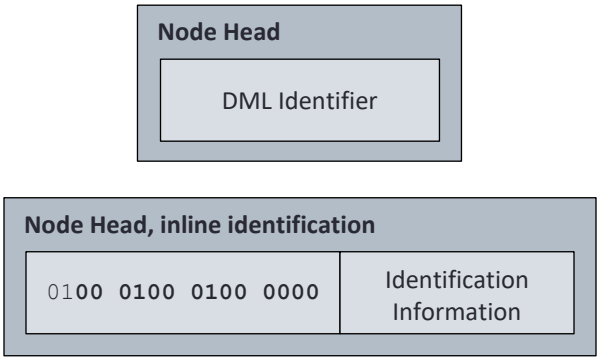
Binary Encoding (Compact-64)	Minimum Value	Maximum Value
1xxx xxxx	0x00	0x7F
01xx xxxx xxxx xxxx	0x80	0x3FFF
001x xxxx xxxx xxxx xxxx xxxx	0x4000	0x1FFFFF
0001 xxxx xxxx xxxx … xxxx xxxx	0x200000	0x0FFFFFFF
0000 1xxx xxxx xxxx … xxxx xxxx	0x10000000	0x07FFFFFFF
0000 01xx xxxx xxxx … xxxx xxxx	0x0800000000	0x03FFFFFFFFF
0000 001x xxxx xxxx … xxxx xxxx	0x040000000000	0x01FFFFFFFFFFF
0000 0001 xxxx xxxx … xxxx xxxx	0x02000000000000	0x00FFFFFFFFFFFFF
0000 0000 xxxx xxxx … xxxx xxxx	0x0100000000000000	0xFFFFFFFFFFFFFFFF

NODE ENCODING

All nodes begin with a DML Identifier, which is a Compact-32. The 32-bit value provides the identifier (although most commonly represented in 7-bits.) Some nodes (called markers) consist only of a DML Identifier. In addition, a specific DML Identifier value is reserved to indicate inline identification. These and other reserved DML Identifiers are built-in to DML, and are detailed in the "Defined Values" section and discussed throughout. All DML Identifiers not reserved for built-in purposes are available for use in defining extensible nodes.

The DML Identifier allows a DML reader to perform a lookup into the currently active translation. From the translation, it can gather the name and type information about the node. The special case of inline identification provides an alternative discussed in the 'Inline Identification Encoding' section.

The DML Identifier and optional identification information constitute the "Node Head". The two possible node head encodings are:



MARKERS

Markers are single-byte nodes, consisting only of their 7-bit DML Identifier. There are three markers found in the basic DML structure: Padding byte, End-Attributes, and End-Container. The markers are recognized by reserved DML Identifier values given in the "Defined Values" section. Padding is to be discarded by DML readers.

NODE ENCODING EXAMPLE

By way of example, a node might be encoded as follows. Each value [xx] represents a byte in the same left-to-right sequence that would be read from the stream. The values are given in hexadecimal. Spaces demark the separation between one sequence and the next but do not represent any actual space in the file.

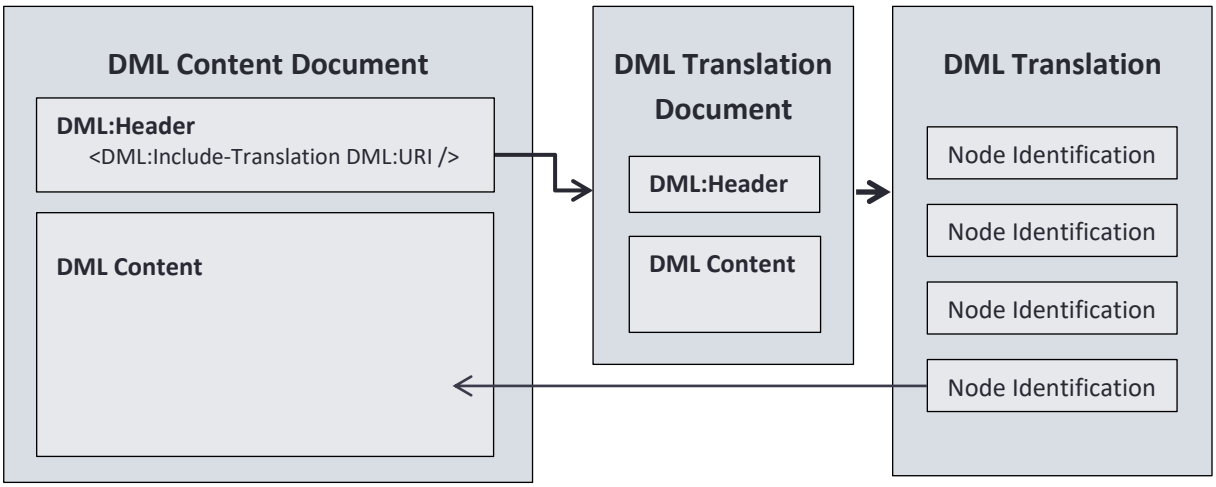
```
Identifier | Data Size | Data Content

 [42] [34]     [82]        [48] [69]
```

This encoding would represent:

- The first byte, [42], represents binary 0100 0010. The '01' prefix indicates that this Compact-32 value includes one additional byte.
- The first and second byte [42] [34] form the complete Compact-32 Value.
    - The binary representation is 01000010 00110100.
    - The '01' prefixed template is `01xx xxxx xxxx xxxx`.
    - The DML Identifier is the "x" value of 0x234.
- From the DML Identifier and translation (not shown), we recognize this node as a string primitive. String primitives begin with a data size, written as a Compact-64.
- The data size, [82], represents binary 1000 0010. The '1' prefix indicates that this is the only byte in the data size. The remaining 0x02 indicates that the data content has a length of 2 bytes.
- The data values [48] [69] correspond to the UTF-8 string "Hi".

TRANSLATION INFORMATION



A DML Translation provides identification and characteristics of nodes. Every node must be identified to be parsed and interpreted. Identification information is encoded either as a DML Identifier value (usually 8-bits) or as inline identification information. When a DML Identifier is used, the DML reader must look up the identification information from a table called a "translation". The translation is specified in the DML Header, which can also reference ("include") other translation documents. These translation documents can be specified either in XML or DML. A translation document can be pre-processed or cached for efficient processing.

In some applications, the translation document and/or DML Header will be hard-coded (called an implied header). The use of an implied DML translation is useful in cases such as a hardware implementation of a communications stream, where a translation document can be descriptive of the implementation and facilitate the use of general-purpose DML readers and writers on a host that must communicate with the hardware.

The translation provides a more complete description of the DML stream. In particular, the DML Translation will indicate:

- The text name for any DML Identifier, which should be compatible with XML naming conventions.
- The type of the DML node.
- Local translations within containers that can redefine DML Identifiers by context.

Rules for encoding and decoding the data content are given by the type of node. DML specifies rules for encoding containers, markers, and the base primitives. Additional primitives may also be used as described in the "Data Markup Language – Primitives" specification. Further details are given in the DML Header section and in the "Data Markup Language – Translation Documents" specification.

INLINE IDENTIFICATION ENCODING

Inline identification of a node occurs when the node's DML ID matches the special inline identification reserved value (given in the Defined Values section). Encoding follows a special sequence outlined as follows:

1. DML Identifier (Compact-32 containing the special inline identification value)
2. Name String Length (Compact-32)
3. Name String (UTF-8)
4. Type String Length (Compact-32)
5. Type String (UTF-8)

The name string should be compatible with XML Naming conventions for elements and attributes.

The type string may consist of any of the type values given in the "Data Markup Language – Primitives" specification or the special value "container". As with the identification information retrieved from a translation document, rules for encoding and decoding the remainder of the node are given by the type of the node.

BASIC STRUCTURE

DML nodes must follow some grammar (sequencing) rules. Container elements begin the sequence, then attribute nodes are included and terminated with an End-Attributes marker. Additional elements are contained next, until the End-Container marker is reached. Reserved space and padding nodes may also occur.

ATTRIBUTES

An attribute is a primitive node that appears before the End-Attributes marker in its container. Thus, attributes always precede elements within their container. An attribute of a particular name may only appear once within a particular container. Attributes may only be of a primitive type. Padding and comments may not appear in the attributes section.

Attributes are attached to their container element. An example container element with attributes written in XML looks like:

```
<Example
      Purpose="Demonstration"
      Utility="DML" Version="1"
      Date="2012-01-21T09:15:00.000000000Z" />
```

ELEMENT TYPES

Elements are different from attributes in that they may be repeated multiple times within their container and come after the End-Attributes marker (but before the End-Container marker). Elements come in two flavors: Containers and primitives. Primitives utilize identical encoding in both elements and attributes.

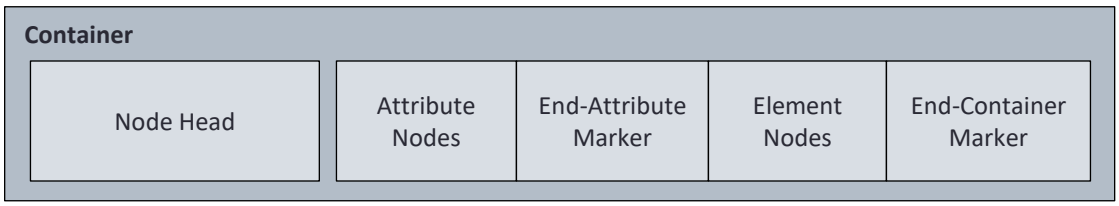
The content of a container consists of attributes and elements. A container is itself an element, permitting a hierarchical tree structure. A container element can be empty, but it must still be included in the sequence (as empty containers can carry information by their presence, multiplicity, and/or sequencing.)

Use cases have differing requirements for the ability to "seek" or cross-reference through a large dataset. These issues are not addressed directly by the basic DML structure, but DML provides all the basic facilities necessary such that proper translation, schema, and software design can accomplish incorporate them. Since DML is a context-sensitive technique, seeks must have a priori knowledge of the context to which the parser is moving. When seeking, cross-referencing, or indexing is necessary, it is strongly recommended that all references be made to the beginning of elements so as to avoid issues with closure, state, or context. The DML:ContentSize attribute is defined to assist with these tasks, but additional definitions are necessary for different applications.

All containers must include a single-byte End-Container marker, even when empty. Containers typically also include a single-byte End-Attributes marker as outlined below.

CONTAINER ELEMENTS

Containers consist of a node head, attributes, elements, and markers for separating them. There is a long and short form for the encoding of containers. The long-form encoding pattern for a container is:

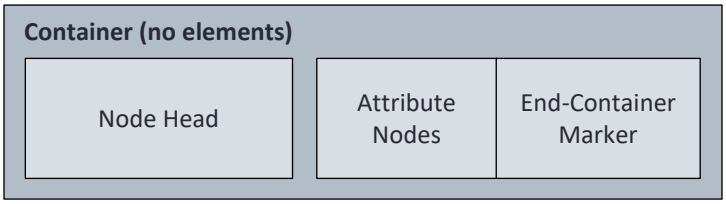


When the End-Container node is encountered, the parser recognizes closure on the current container element. This is the XML equivalent of a closing tag. The End-Container marker *must* appear exactly once for each container element and *must not* appear for any other kind of node.

CONTAINER ELEMENTS WITH NO CHILDREN

A container element that contains no element nodes can be encoded in a shortened form. DML Readers *must* recognize the shortened form, although writers may utilize either form.

The End-Attributes Marker and Element Nodes are omitted. The reader detects the End-Container Marker instead of an End-Attributes Marker and recognizes that the container had no Element Nodes. Attribute Nodes may be present ahead of the marker. A completely empty container element can be encoded in as few as 2 bytes.



CONTENT SIZE

Any container may contain the "DML:ContentSize" attribute, which is an unsigned integer primitive. When specified, the ContentSize *must* give the exact size, in bytes, of the element nodes of the container. This value should be equal to the number of bytes contained between the End-Attributes and End-Container markers of the container (excluding the End-Attributes and End-Container markers themselves). A generic DML parser can use the ContentSize attribute, when present, to accomplish partial loading of large nodes or to rapidly seek through large files.

The schema for a file format should give consideration as to whether the DML:ContentSize attribute is required on certain containers. In order to calculate the DML:ContentSize value, the entirety of the container must be ready to encode beforehand. It is therefore not suitable to apply it to the top-level of documents that will be used for streaming. However, it is ideal for individual containers within a streaming document. For example, a live stream of video cannot use a ContentSize at the top-level (because the size is unknown until completion) but should use a ContentSize around each frame to facilitate seeking between frames.

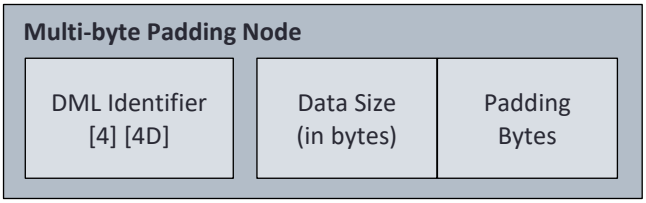
The ContentSize does not include information about the size of attributes of the container, nor is its sequence within the container's attributes specified. File format design should take these constraints into consideration in order to provide an accessible hierarchy. For this reason, it is recommended that large data be kept in element primitives, not attributes. Alternatively, large or numerous data attributes can be wrapped by a higher-level container that contains the ContentSize attribute to permit skipping over the inner container.

RESERVED SPACE AND PADDING

Reserved space or space needed for alignment purposes can be marked with "Padding" nodes. No padding is required in DML, but is available as a tool for DML writers. A DML reader will usually discard any padding that it encounters. Padding is available in three forms:

- Single-byte padding marker
- Multi-byte padding node
- Comment node

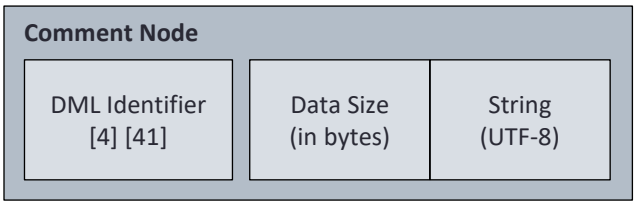
For single or small runs of padding (usually 10 bytes or less), the single-byte padding marker can be used. The single-byte padding marker is encoded as the DML Identifier value 0x7D (when encoded as a Compact-32, this is a 0xFD byte). For larger padding, the multi-byte padding node should be utilized (DML ID of 0x44D). The multi-byte padding node is encoded as follows:



The data size is encoded as a Compact-64 value. Since the multi-byte padding node includes an arbitrary data size length, the parser can discard the entire block at once.

Padding can be used to achieve a particular alignment, or can be used to reserve space in a stream for later overwriting. Padding must occur as a valid freestanding node (i.e. it will not be recognized in the middle of a byte array data sequence).

Readers *may* also treat comment nodes as padding and ignore or discard them. If converting to/from XML, the DML reader *should* include comments. Comment nodes use the same encoding as string primitives:



The data size is encoded as a Compact-64 value. Padding and comments may appear only as elements – not as attribute nodes.

DOCUMENT STRUCTURE

The document structure of any DML file or stream is built upon the node encoding and basic structure described above. The DML structure provides a minimum description to provide uniform and well-behaved parsing by a reader. The remaining structure can be composed using the container, element, attribute, and translation mechanisms described herein.

DML can be utilized in two ways: documents or fragments. A document is the complete way of utilizing DML while a fragment is a non-compliant but common use case. A document is a fully-described document, whereas fragments are especially useful in communications channels where both endpoints have implicit information about the protocol being employed and individual messages need not be fully described.

Any DML document consists of two parts:

- DML Header
- DML Body

The DML Header is a single container with name “DML:Header” and content to describe the process of reading and interpreting the body. All nodes of the DML Header are defined herein and it is not directly extensible. A special translation space, the TSL2 translation is employed during Header parsing, and this can be represented as a contextual translation to be employed whenever the DML:Header container is being parsed – consistent with other DML parsing if the translations are defined properly. A translation that provides this behavior is given in the Defined Values section.

The DML Body is a second container within the file or stream that immediately follows termination of the DML Header container. The DML Body provides the top-level container for the document or stream. The DML Header can be thought of as a processing directive, similar to the processing directive that begins most XML files. The DML Body is completely extensible.

DML Fragments contain one or more DML containers but omits the DML Header. To use a DML Fragment, the DML Header must be implied and known a priori. In some applications, it may be useful for one end (i.e. hardware device) of a communications or storage path to hardcode the behavior of the DML Header while the other side (i.e. a host PC) can utilize a full DML parser. In this case, the host PC can use an implied header that describes the hardware’s communications channel in DML.

The basic structure is accomplished by means of a few reserved DML Identifiers (given in the Defined Values section) that have special meanings at the parser level.

DML HEADER

The DML Header includes nodes defined for all DML readers that can be used to identify or verify the DML version and required reader version, provide the translation for the DML Body, and enumerate any primitives and codecs required for decoding.

Upon detection of the DML:Header container, the DML reader must begin recognizing the TSL2 translation and using it to define the translation for the document’s body. TSL2 is also known as the Translation Language of DML. The translation language provides a DML:Include-Translation container that can be utilized to reference an external URI containing either an XML or DML Translation Document. If a DML document is specified, then this document contains its own DML:Header and its body contains a DML:Translation top-level container. The contents of the DML:Translation are logically integrated in-place of the include statement in the original document’s header. If a XML document is specified, then the document contains a DML:Translation root node and must utilize the XML representation of the TSL2 translation. An XML document may contain the XML preprocessing directive, but no DML:Header will be incorporated in that document.

All DML readers must recognize the small base set of primitives. Some applications may require additional primitives, and the primitive include statement is designed to enable this. Primitive include statements can be placed either in the DML Header or any Translation Documents that are themselves referenced by the DML Header. If inline identification will be used with types that require additional primitive(s), then those primitive(s) must be referenced in the DML Header. See the “Data Markup Language – Primitives” specification for more information.

Readers *should* disregard duplicate primitive include statements. Primitive include statements are processed in the order they are encountered, so the content stream's DML Header has final precedence on codec choice.

See the "Data Markup Language – Translation Document" specification for more information and a formal description of the DML:Header, DML Translation Documents, and translation language. It is recommended that translations be fully contained within a DML Translation Document and that the DML:Header consist of only a DML:Include-Translation statement referencing the DML Translation Document. In recommended usage, the DML:Header may also override or further specify primitive codecs to be utilized after the DML:Include-Translation of the translation document.

DML BODY

The DML Body provides the top-level container for the document or stream following the DML Header. Unlike the DML Header that always has the name "DML:Header", there is no fixed name for the top-level container of the body. The name of the DML Body's container element can be considered equivalent to the XML root-node of a document.

It is suggested that DML:Version and DML:ReadVersion attributes be present on the body container as well as on the DML Header container. On the DML header container they represent the required DML parsing version in order to understand the encoding of the document or stream. On the body container, they represent the version of the file format definition being utilized, and the necessary reader version to be able to understand the body content.

The translation also defines an optional container element called "XMLRoot". This element is only utilized for conversion of the DML document to an XML equivalent, and provides any attributes that should be appended to the XML root node (corresponding to the DML Body container element) during conversion. It can be used to attach XML schema or namespace references when using a particular DML translation. These definitions *may* be incorporated directly to the DML Body container, but placing them in a Translation Document can result in a more compact data file since these definitions are typically fixed for a specific Translation Document and need not be repeated in every content document. References to schemas and namespaces are not required for use of DML and have no effect at the DML parser level.

DEFINED VALUES

This section provides the predefined numbers utilized by DML. All DML Identifiers are given in hexadecimal, with the compact-prefix omitted. For example, DML ID [4][40] would be encoded as binary `01`**`00 0100 0100 0000`**.

Name	DML ID	Type	Description
DML Structural			
DML:InlineIdentification	[4][40]	Special	See Inline Identification section for discussion.
DML:Comment	[4][41]	Special	Comment. Content should be disregarded unless being converted to XML, where it should represent an XML comment.
DML:Padding	[4][42]	Special	See Padding section for discussion.
DML:PaddingByte	[7D]	Special	See Padding section for discussion.
DML:EndAttributes	[7E]	Special	See Containers section for discussion.
DML:EndContainer	[7F]	Special	See Containers section for discussion.
DML Standardization/Optimization			
DML:ContentSize	[7C]	UInt attr	See Containers section for discussion.
XML:CDATA	[7B]	String	Facilitates conversion of text data to/from XML. See "Data Markup Language – XML Conversion" specification.
Multipurpose Attributes			
DML:Version	[4][50]	UInt attr	The version of the DML writer that created the file. Can be attached to DML:Header, body top-level container, and other uses.
DML:ReadVersion	[4][51]	UInt attr	Minimum DML version that a reader must understand to interpret this file. If not provided, Version is considered instead.
DML Header			
DML:Header	[4][44][D4][C2]	Container	Contains the DML Header. All DML documents begin with this container. This container employs the TSL2 translation described in the "Data Markup Language – Translation Documents" specification.
DML:DocType	[4][52]	String attr	Identifying description of the file format used, i.e. "BrandName-Video-Container"
See the "Data Markup Language – Translation Documents" specification for additional pre-defined values that apply to both the DML:Header and DML:Translation containers.			

BUILT-IN TRANSLATION DATA

The associations described in this document must be “built-in” to any DML translator. Additional translation definitions should be merged with this built-in set at their top-level. The built-in set is summarized in XML as:

```
<DML:Translation DML:URN=”urn:dml:dml3”>
       <Node id=”1104” name=”DML:Version” type=”uint” usage=”attribute” />
       <Node id=”1105” name=”DML:ReadVersion” type=”uint” usage=”attribute” />
       <Node id=”1106” name=”DML:DocType” type=”string” usage=”attribute” />

       <Container id=”71619778” name=”DML:Header”>
              <DML:Include-Translation
                     DML:URI=”urn:dml:tsl2” />
       </Container>

       <Node id=”123” name=”XML:CDATA” type=”string” />
       <Node id=”124” name=”DML:ContentSize” type=”uint” usage=”attribute” />

       <!-- The following additional definitions exist, but should
              be interpreted before the properties are utilized. -->
       <Node id=”1088” name=”DML:InlineIdentification” type=”special” />
       <Node id=”1089” name=”DML:Comment” type=”special” />
       <Node id=”1090” name=”DML:Padding” type=”special” />
       <Node id=”125” name=”DML:PaddingByte” type=”special” />
       <Node id=”126” name=”DML:EndAttributes” type=”special” />
       <Node id=”127” name=”DML:EndContainer” type=”special” />
</DML:Translation>
```

A second built-in translation set is also defined that is applicable when its translation URN is specified, and is utilized for translation documents. See the “Data Markup Language - Translation Document” specification for this additional translation definition and explanation.

All DML IDs used at the top-level of DML3 are reserved and *must not* be reassigned in local translations. Dml reader and writers should generate an error upon any attempt to do so.