DATA MARKUP LANGUAGE PRIMER

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

Version: V3.1

CONTENTS

INTRODUCTION

This document provides a brief introduction to the Data Markup Language. For a more detailed review and references, see the specification documents:

- Data Markup Language: Specification describes the fundamental structure and language.
- Primitives: Specification describes primitive support and encoding in DML.
- Translation Document: Specification describes the translation language.
- DML Encryption and Compression: Specification of DML standardized compression and encryption.
- Relationship to other Encodings: Discussion of DML relative to other encodings and translations between them.

Data Markup Language (DML) is a simple but versatile means of storing binary data of arbitrary content and shape. It is capable of supporting virtually any data that a computer would store and understand in a single, unified file-format or stream encoding.

DML is strongly related to the Extensible Markup Language (XML). The reader should be familiar with XML basic structure before attempting to understand DML, and a quick overview of the relevant concepts is given in the next section. XML Schemas, namespaces, and types are not important to understanding DML but the elementary syntax is used throughout and provides the design principles behind DML. The basic structure of DML matches the basic structure of XML, so all examples presented herein will use XML as a pseudocode language.

The fundamental unit of encoding in DML is a "node". Nodes include elements, attributes, and some special cases. Containers are one kind of element, and containers can include other elements and attributes. This permits a tree (hierarchal) structure where information can be contained within one another.

Encoding of DML can include a wide variety of topics and formats, but the fundamental language is comprised mostly of just two: The Compact integer format and UTF-8 strings. That is not to say that DML is limited to storing information in those two formats – in fact just the opposite. Those two encodings simply form the basic structure on which more complicated structures are built.

DML is designed to handle as many use cases as possible, hopefully all of them. That requires a rich and large structure to support, and DML is designed to have the options available when you run into a wall and discover you need a new capability. That leads to there being a lot of details in DML that aren't terribly important to get started. So to begin, we will focus on the simple case of "inline identification" and ignore topics such as translations.

XML QUICK TUTORIAL

Extensible Markup Language (XML) includes a powerful and rich feature set that spans many specifications and documents, but here we care only about the most basic structure. In particular, XML includes three kinds of "tags":

<Name>	Opening Tag
</Name>	Closing Tag
<Name />	Combined Opening-Closing Tag

Each opening + closing tag pair forms what is termed an “element”. The shorthand, combined open-close tag also defines a single element, but the element has no content inside it.

Elements can include attributes. Attributes are name-value pairs that are included on the tag. Both the name of the element and the attributes are arbitrary, which is where the extensibility comes in.

Element and attribute names may not include whitespace, but whitespace is otherwise ignored in XML. Consider a short XML example:

```
<?xml version=”1.0” encoding=”UTF-8” ?>
<Library Location=”Old Main”>
      <Book Title=”To Kill a Mockingbird” Author=”Harper Lee” />
      <Book Title=”The Hobbit” Author=”JRR Tolkien” />
      <Book Title=”War and Peace” Author=”Leo Tolstoy” />
      <Periodical Title=”Popular Science”>
            <Edition Date=”January 2012” />
            <Edition Date=”February 2012” />
            <Edition Date=”March 2012” />
      </Periodical>
      <Book Title=”Of Mice and Men” Author=”John Steinbeck” />
</Library>
```

What is the out-of-place looking <?xml…> marker? That is called the XML declaration or a processing directive. It tells a computer reading it that the document is XML and that it is encoded in UTF-8 (Unicode) format.

The rest of the document forms a tree structure. The books are all contained in the library. The periodical is also included in the library, and 3 editions are included in the periodical. Each book and periodical has several attributes attached to it – such as the title and author. Whitespace is arbitrary, but is laid out for easy reading.

ENCODING

The basic logical unit in DML is the node and the basic encoding tool is Compact numbers. They allow us to incorporate an integer value in the smallest representation possible while keeping to 8-bit boundaries. They use a structure very similar to the encoding of UTF characters where the first bit(s) give you an indication of how many bits of information follow. Compact numbers up to a value of 127 can be encoded using 8-bits, and so DML is designed to use numbers 0 to 127 almost exclusively with the capacity for larger numbers available.

COMPACT UNSIGNED INTEGERS

The compact unsigned integer is a variable-length encoding similar to the UTF-8 encoding system used for strings. 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

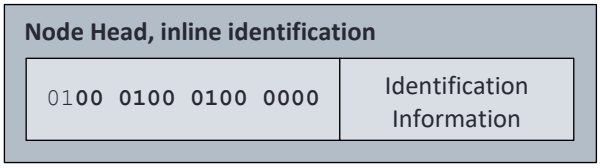
The smallest encoding for the unsigned integer value should always be utilized. When read, the value is zero padded back into a 32-bit unsigned integer in memory. When multiple bytes are used, a big-endian sequence is

employed. We refer to the above encoding as a Compact-32 value. Data Markup Language also makes use of a Compact-64 value that follows a similar pattern (identical through the 0001 prefix):

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	0xFFFFFFFFFFFFFFF

NODE ENCODING

The full discussion of encoding nodes is most of what DML is about and is a large topic, but the simple cases cover most needs. I'll begin with the inline identification version of a node that gives you a lot of flexibility to begin with.



Why the long constant for every node? Well, inline identification is not the most efficient storage mechanism that DML has. In fact it is wasteful. But, it is simple and extensible. More compact options are waiting in DML when you need them.

That constant is actually a Compact-64 value. It is a particular value, which is what tells the DML Reader that this node is using inline identification.

The Identification Information contains the following and in this sequence:

1. Name String Length (Compact-32)
2. Name String (UTF-8)
3. Type String Length (Compact-32)
4. Type String (UTF-8)

The node's name is similar to XML. In the XML example above it might be "Library" or "Location". It can include dashes and is case-sensitive. It should match XML naming conventions.

The type string is a bit harder to explain. One case is for the type to be "container". This would indicate that the node is an element that contains other nodes and follows a container template for its encoding format, discussed later. All of the tags in the Library example except the special <?xml?> directive were containers, even the books that contain no child elements. If the node is not a container then it is either a primitive or among a few special markers that we'll get to later. We can categorize all of DML this way:

- DML Nodes
    - Elements
        - Containers

        - Element Primitives
    - Attribute Primitives
    - Special Markers
    - Comments/Padding

We already stated that nodes are the fundamental unit in DML, so why primitives? The simple analogy is that a primitive specifies the type such as integers, floating-point, string, etc. while nodes give instances of these types. In a programming language, it's akin to declaring "int i". The type is integer (primitive) and the variable is i (node).

There are an unlimited number of primitive types, but until you request them you have only 3 available: "uint", "string", and "array-U8". If the only primitive type available was "string", then DML would be very similar to XML though a bit more compact. We've seen lots of software attempt to pack information into XML's strings and while it works, the encoding and parsing are inefficient and cumbersome.

You can request more primitives by adding a line to your DML header. There are a plethora of additional primitives available: Signed integers, Booleans, Floating-point, date & time, base-10 floating point, arrays, matrices, and even encrypted and compressed primitives. In the spirit of DML, if those types aren't enough or aren't in an encoding style that suits your application then you can define your own and insert your own software to encode/decode it as an extension.

That's enough to give a basic idea of what a node looks like. Let's see an example.

NODE ENCODING EXAMPLE

Each value [xx] represents a byte in the same sequence that would be read from the stream. The values are given in hexadecimal. The example stream:

```
44 40 88 4c 6f 63 61 74 69 6f 6e 86 73 74 72 69 6e 67 88 4f 6c 64 20 4d 61 69 6e
```

Breaks down as:

DML Identifier	[44] [40]
Name String Length	[88]
Name String	[4c] [6f] [63] [61] [74] [69] [6f] [6e]
Type String Length	[86]
Type String	[73] [74] [72] [69] [6e] [67]
String Primitive Length	[88]
String Primitive	[4f] [6c] [64] [20] [4d] [61] [69] [6e]

This encoding would represent:

- The first byte, [44], represents binary 0100 0100. The '01' prefix indicates that this Compact-32 value includes one additional byte.
- The first and second byte [44] [40] form the complete Compact-32 Value.
    - The binary representation is 01000100 01000000.
    - The '01' prefixed template is `01xx xxxx xxxx xxxx`.
    - The DML Identifier is the "x" value of 0x440.
- This DML Identifier constant always indicates inline identification. We know identification information follows.

- The name string length (a Compact-32) is a single byte, [88]. This has template 1xxx xxxx and the "x" value is 8, thus our name string is 8-bytes.
- The name string is contained in the next 8-bytes, which are the UTF-8 encoded string "Location".
- The type string length ([86]) is another Compact-32, this time indicating a 6-byte string.
- The 6 bytes which follow are the UTF-8 encoded string "string", the primitive type.
- The string primitive consists of the same pattern we've just seen: A Compact-32 giving the length followed by the string itself.
    - The string length byte ([88]) indicates an 8-byte string.
    - The 8-byte string follows and is UTF-8 encoded "Old Main".

NODE IDENTIFICATION

We've seen node encoding when inline identification is involved. When that first DML Identifier (DMLID) constant is not the value I've shown above, the DML reader will figure out the identification from the DMLID itself instead of looking for the name and type strings. In the example above, the inline identification told us that the node was named "Location" and had type "string". If a specific DMLID were used instead, the DML reader should know the node's name and type just from that DML Identifier.

But how would that be extensible? In general, the key to it being extensible is that the DML Identifier goes into a look-up table called the DML Translation. The lookup table contains all the information that was previously given in inline identification.

Why support inline identification if we have the DML Translation? After all, the DML Translation is more space efficient, right? Well, 2 reasons. First, the DML Translation comes in a second file. For many simple tasks a self-contained format is preferable. Secondly, even if a DML Translation is in use, we still need to allow mixed-in extensibility. For example, imagine a video file format written in DML. A large and complex translation probably specifies all the nodes you'll need to encode standard video data. But, let's say you are designing a special camera product and it is important in your application to record the GPS coordinates where each video frame was recorded. The video file format includes nothing for GPS data. In this case, you can use add a string attribute to each <Frame> container using inline identification that incorporates your GPS data. A lot of software won't know what to do with it, but it follows DML so they just can ignore it and get to playing back the video. Meanwhile, your custom software has the GPS data it needs nicely packed in.

As you can see in figure 1, use of inline identification just means we're inserting some extra information that would otherwise come from the lookup table through the DML Identifier number. DML Translations are a more advanced topic, but there are a few pre-defined identifiers that work in any DML. In fact there's a "base" DML Translation called DML3 that offers these pre-defined identifiers and must be supported by all DML readers.

The use of a DML Identifier (DMLID) and DML Translation can permit very tightly packed data format. The Compact-32 encoding means that most node identifiers can be represented in a single byte, but nearly a billion identifiers are ultimately available. DML also supports the use of "container context" that means that the DML Identifier must be considered in context. This adds complexity to the parser, but permits even more than 128 values to be available for use within single byte DML Identifiers without using any additional space in the file or stream.

Lastly, I'll mention that DML also lends itself to resource-tight environments. It might not seem so with all the complexity that is involved, but it turns out that most DML translations can easily be hard-coded into software or

firmware. In fact, it can be hard-coded in very easily when a single, specific DML format is all that must be supported. Conveniently, connections to a PC can use a full DML parser just given a translation document to teach them the device's language.

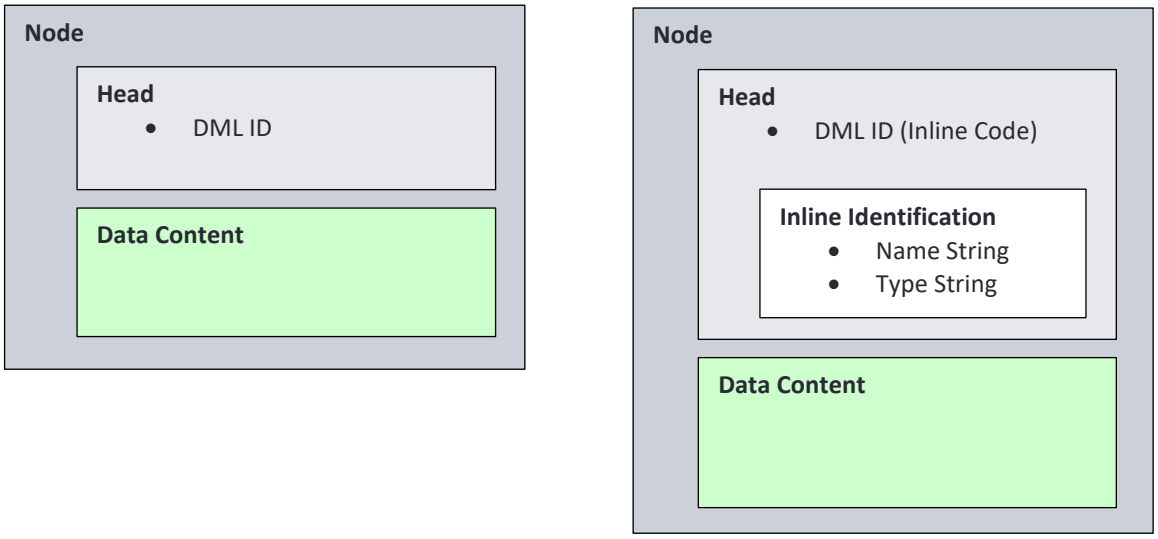


Figure 1. Ordinary nodes (left) and nodes with inline encoding (right).

DML STRUCTURE

DML consists of containers, attributes, and elements. Containers are themselves elements, which permits a recursive tree structure. Elements can also be primitives. Attributes can only consist of primitives. This is all nearly identical to the structure found in XML.

When a node is a primitive, it contains some basic value, such as an integer. Primitives come in two flavors: elements and attributes. The only difference is grammatical: all attributes for a container must be provided before any elements, and a particular attribute can only appear once in a container. Containers usually have a single-byte End-Attributes marker node in them that separates the attributes from the elements.

Why the distinction between elements and attributes? Attributes are a helpful tool for gathering the set of information that is required in order to characterize what comes next. A simple example is an image. You need to know some information about the image in order to interpret the image – such as whether it is a color (3 channel) or greyscale (1 channel) image. Since this information is required in order to understand the information that comes next, it should be incorporated as an attribute – before the other information.

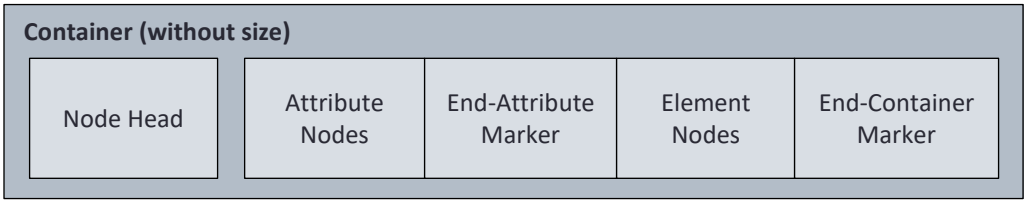
A key difference between DML and XML is the availability of primitives and primitive encoding. In XML, one could say there are a few basic primitives available such as integers, dates, etc., but there is certainly only one primitive encoding - text. In DML, the primitives provide instructions for encoding the information in a representation that is less human-friendly but more computer-friendly. We'll revisit primitives again later.

DML containers can contain additional DML containers. This provides the "tree" representation that DML and XML both use. While the shape of the tree and the data being represented depends on the file format and data being

used, the DML specification does require at least two containers in each DML file. Thus, each DML file has a little bit of overall structure to it. We'll revisit the overall DML file structure again later, but first let's look at how we encode the hierarchal structure.

CONTAINER ENCODING

Containers begin with a node head just like any other node. They can use inline identification, in which case they have the type string "container". Following the node head is any number of attribute nodes. This is usually followed by an End-Attributes marker node and then any number of element nodes follows. Finally, all containers include a single End-Container marker node.



I use the word "usually" on the End-Attributes Marker. There is only one exception: if there are no element nodes than the container can also omit the End-Attribute Marker and go straight to the End-Container Marker. The reader will see the End-Container Marker and know that this meant there were no element nodes.

DML FILE STRUCTURE

As mentioned earlier, all DML Documents have two top-level containers. The first, called the DML Header, is similar to an XML file's processing directive – the one that tells an XML reader what XML version and text encoding is used. The second is the top-level container of your data, called the DML Body.

In some applications, the DML Header and the top-level container can be implied instead of actually stored or transmitted, such as over a communications channel where both sides know the protocol in advance. We call these DML Fragments. DML Fragment can be thought of as a DML Document where we only exchange what happens inside the DML Body and everything else is implied. Here we will focus on DML Documents.

DML HEADER

The first container is the DML:Header container, providing information about what the file is about and where to find the necessary information to process it. The DML:Header container node is always the first thing in a DML document – which means that a DML reader can check the first 4 bytes against the DML:Header pattern to verify that the file is actually DML. If you are using only inline identification inside your document body, then the header can be as simple as:

```
<DML:Header />
```

If a DML Translation is used for the file, then it might look like this instead:

```
<DML:Header>
      <DML:Include-Translation
            DML:URI="http://www.dmlexamples.org/example.dml" />
```

```
</DML:Header>
```

Primitive sets are a key tool in DML that will be discussed more later, but for now let's see how we request them in either your DML Translation or your DML:Header. For example, if you need to encode floating point numbers in little endian format, which are given by the "common" set as:

```
<DML:Header>
      <DML:Include-Primitives DML:Set="common" DML:Codec="le" />
</DML:Header>
```

There are several other attributes that can be attached to the DML:Header. The rest are all optional, but each serves a useful purpose. A quick overview of the things you can attach to the DML:Header:

- DML:DocType is a particularly good attribute to include. The DML:DocType can provide a quick lookup of the file format without needing the DML Parser to process the full DML Header or DML Translations. The DocType is optional and the DML Body container name can also identify the file content. The recommended practice is that the DocType match the DML Body container name, but is a string.
- Version can be included in order to tell the DML reader what version of DML you are using.
- ReadVersion can be included to be more specific and tell the DML reader what version of DML *it* needs to be using (minimum).

A more full-featured header example is given:

```
<DML:Header DML:DocType="Video" DML:ReadVersion="3">
      <DML:Include-Translation
            DML:URI="http://www.dmlexamples.org/VideoExample.dml"
            DML:URN="urn:uuid:6e8bc430-9c3a-11d9-9669-0800200c9a66"
            />
      <DML:Include DML:Primitives="arrays" DML:Codec="be" />
</DML:Header>
```

The DML:Header can be thought of as the DML version of the XML processing directive. Following the DML:Header is the top-level container for the document.

DML BODY

The name of the top-level (body) container is up to the file format, and is usually the key indicator as to what kind of file it is. The top-level body can be encoded either with a DML ID, defined in the DML:Header or its included translations, or it can be encoded with inline identification.

DML documents, like XML files, contain only one top-level container. However, that top-level container can contain anything you want (including more containers) and as many repetitions as you want (at least for elements).

Now our first complete example DML document is presented in XML pseudo-code:

```
<DML:Header DML:DocType="Library">
      <DML:Include-Translation
            DML:URI="http://www.dmlexamples.org/LibraryDirectory.dml"
            DML:URN="urn:uuid:6e8bc430-9c3a-11d9-9444-0800200c9a66"
      />
</DML:Header>
<Library Location="Old Main">
      <Book Title="To Kill a Mockingbird" Author="Harper Lee" />
      <Book Title="The Hobbit" Author="JRR Tolkien" />
      <Book Title="War and Peace" Author="Leo Tolstoy" />
      <Periodical Title="Popular Science">
            <Edition Date="January 2012" />
            <Edition Date="February 2012" />
            <Edition Date="March 2012" />
      </Periodical>
      <Book Title="Of Mice and Men" Author="John Steinbeck" />
</Library>
```

EXCEPTIONS TO THE PATTERNS

DML contains a few exceptions. Inline identification could be considered one exception, as already discussed. The marker nodes, padding, and comments are structural and do not classify as either attributes or elements. Finally, when using DML Translations, we define short identifiers in order to make our data storage really efficient.

MARKERS

Markers are simple single-byte nodes present for structural purposes only. The finding of a reserved DML Identifier itself is sufficient to accomplish the entire purpose of the node – to mark the structure. The three built-in single-byte marker nodes are: Single-byte padding, End-Attributes marker, and End-Container marker.

PADDING

Padding is needed for various reasons, most commonly because a writer must reserve space at the beginning of a file and then rewrite it later. A DML reader should simply ignore and discard padding.

In DML, padding comes in 3 flavors. The first is the single-byte padding that can be immediately discarded when read. When the DML reader is reading along and finds that the first byte of a node (DMLID) is 0xFC, the DML reader knows we have a single-byte padding. It discards the byte and looks at the next one to start another node. A DML writer should avoid writing more than about 10 consecutive single-byte paddings.

The second flavor of padding is the full padding node. This has a DMLID of 0x442, which becomes 0x4442 when it is encoded as a 2-byte Compact-32. After the DMLID, we look for a data size indicator, a Compact-64, immediately

following the node head. The data size indicator tells us how many bytes of reserved space (padding) should be skipped. A DML writer might use the full padding node to reserve a larger bit of space on disk for later rewriting, say 64KB for example. If it doesn't use the 64KB later, it may need to write out another padding node to continue marking some unused space.

The last flavor of padding is comments. These exist so that DML can be converted to and from XML, but for most purposes they can be treated just like more padding. Comments come in a node with DML Identifier 0x441 and have the same format as a string primitive.

SHORT IDENTIFIERS

There is one last exception in DML applicable to the DML Identifiers that we read from the file. DML Identifiers fit into a Compact-32 value, which provides 7-bits of identification for a single-byte representation. Thus we have only 128 identifiers available if we want to maintain compactness. A few of those are reserved for DML special purposes, so 124 are available. This is plenty to represent most needs, but DML also offers context sensitive identifiers for maximum compactness.

The DML Translation has hierarchy to it. That is, containers can define translations that exist only within that container. This simple trick allows us to utilize an almost infinite number of DML identifiers and keep to 8-bits to start nearly any node.

Consider the following XML example:

```
<Monarchy>
      <Person Title="King" Name="Arthur" Born="498" />
      <Person Title="Queen" Name="Guinevere" Born="499" />
</Monarchy>
```

Let's try assigning IDs to these using local translation context. We'll assign Monarchy the DML Identifier 1. The DML reader recognizes that it is inside of Monarchy when it reads Person, so we can assign 1 again. The DML reader recognizes that it is inside of the Person container and we can assign Title, Name, and Born DML identifiers 1, 2, and 3 respectively. In this way we have assigned 5 different nodes to only 3 DML Identifiers, but the DML reader will have no trouble figuring out what is referenced by each identifier.

Although the DML encoding will see only the DML Identifier, when we are discussing DML we will also need to specify the DML Identifier as being a combination. For example, we could reference 'Born' above as identifier 1:1:3 because it is contained inside DML ID 1 (Monarchy) inside ID 1 (Person) and is identifier 3. It also makes sense to designate the DML ID for 'Born' as Monarchy:Person:3, and you can lookup Monarchy and Person IDs separately.

PRIMITIVE SETS

Primitives include the basic building blocks of data that are allowed as nodes in DML. At a minimum, every DML parser should be able to handle what is called the "base set" of primitives: unsigned integers, strings, and byte arrays.

Many additional primitives are defined, such as signed integers, Booleans, floating-point types, base-10 floating points, arrays, matrices, compression, and encryption. It is also possible to define your own primitives and mark when they are needed in the DML translation, but this should be used only when existing solutions are insufficient.

Historically, a downside of incorporating more primitives into a file format or encoding system is that more complex, compound primitives can have many different representations – often with pros and cons to each. For example, an array of integers might be defined in either little-endian, big-endian, or some kind of endian-neutral format. The little-endian version is ideal for Intel processors, but big-endian performs better on a few embedded processors and is "network format" throughout internet standards. An endian-neutral format seems to avoid these problems, but is less efficient for processing and storage. The DML solution is to define "primitive sets" so that a necessary codec can be selected for each application.

BASIC PRIMITIVES

Primitives are always encoded inside of a node, providing the data content.

Unsigned integers (type="uint") are represented by Compact-64 values already discussed. Strings (type="string") have been addressed in previous examples throughout the primer, but formally they consist of a Compact-64 length indicator followed by the UTF-8 encoded string. Byte arrays (type="array-U8") consists of a Compact-64 size indicator followed by the specified number of bytes.

ADDITIONAL PRIMITIVES

Many additional primitives are available. The basic set was kept minimal because everyone has a different needs for their encodings. Even floating-point values can be encoded in big or little endian and with different configurations.

When using primitives beyond the basics, you must include a note in your DML:Header or DML:Translation to that effect. A DML reader can know up-front what kind of primitives it's going to need to be able to interpret the document. In order to include the little-endian version of the "common primitives set", the following simple DML header might be used:

```
<DML:Header>
      <DML:Include-Primitives DML:Set="common" DML:Codec="le" />
</DML:Header>
```

The extra primitives and the header/translation directive for using them are discussed in more detail in the "Data Markup Language – Primitives" specification.

TRANSLATION DOCUMENTS

When your DML content has outgrown the use of inline identification and you are ready for some really high-efficiency stuff, DML Translation documents are the next step. You may also need to understand them if you are reading someone else's file format.

A DML file containing data of interest is called the DML Content. The DML:Header of your DML Content file can contain <DML:Include-Translation> directives with a Universal Resource Identifier (URI), usually an internet address. This directive and its URI tells you where to locate the DML Translation document, a second file, to be able to understand your DML Content file.

The language of the Translation document's body is the same as the DML:Header language, and both are referred to as translation language. The <DML:Include-Translation /> and <DML:Include-Primitives /> are part of the

translation language, which also tells us that translation documents can reference still more translation documents by more include statements.

The translation document is a separate file or resource from the DML content even though it will be merged into the DML:Header in processing. It can be encoded either in DML or XML and XML is preferable in most cases. DML translation documents use only string and unsigned integer encodings, so they can be fully represented in XML. Writing a translation document and schema can be a useful tool in planning (and sharing) what a file format will look like.

A Universal Resource Identifier (URI) is either an Internet address in the form of a Universal Resource Locator (URL), or it is a Universal Resource Name (URN). If it is a URL, then the software can simply jump on the internet and grab the resource, translate it, and know the translation rules for the DML Content. If it is a URN, then the software can at least uniquely identify the resource and can ask the user to provide a translation document. There are a few special URN values that reference "built-in" translations.

The first built-in URN references the DML3 translation, which must be supported by any compliant DML reader. The URN is "urn:dml:dml3", however this translation need never be explicitly requested – all DML documents must provide this translation as a starting point and append the rest of the translation language.

An additional DML Translation that describes the translation language itself is requested by:

```
<DML:Include-Translation DML:URI="urn:dml:tsl2" />
```

We also request the optional DML Encryption and Compression translation outlined in the "Data Markup Language – DML Encryption and Compression" specification as:

```
<DML:Include-Translation DML:URI="urn:dml:dml-ec2" />
```

Creating translations is examined more carefully in the "Data Markup Language – Translation Document" specification, but an example translation document is provided:

```
<DML:Translation
      DML:URN="6e8bc430-9c3a-11d9-9669-0800200c9a66"
      >
      <DML:Include-Primitives DML:Set="common" DML:Codec="le" />

      <Container id="100" name="Library">
            <Node id="1" name="Location" type="string" />
            <Node id="2" name="Title" type="string" />
            <Node id="3" name="Author" type="string" />
            <Container id="4" name="Book" />
            <Container id="5" name="Periodical">
                  <Container id="1" name="Edition">
                        <Node id="6" name="Date" type="datetime" />
                  </Container>
            </Container>
      </Container>
</DMLTranslation>
```

Another example taken from the Translation Document specification is provided to illustrate the hierarchal format and inclusion of additional translations:

```
<DML:Translation>
      <!-- Information on this translation can be found at [example URL]. -->

      <Container id=”1” name=”Slideshow”>
            <Container id=”1” name=”New-Slide” />
      </Container>
      <Container id=”2” name=”Video”>
            <Attribute id=”1” name=”Style” type=”string” />
            <DML:Include-Translation
                  DML:URI=”http://dmlexamples.org/Video.dml” />
      </Container>
      <Container id=”3” name=”Audio”>
            <Attribute id=”1” name=”Style” type=”uint” />
      </Container>
</DML:Translation>
```

And related DML content as follows:

```
<Slideshow>
      <Audio Style=”15” />
      <New-Slide>
            <Video Style=”raw”><Audio Style=”11” /></Video>
      </New-Slide>
</Slideshow>
```

The algorithm in DML to determine what a DML ID means goes up the *content* tree first, and then goes up the *translation* tree. That is because not every container has a local translation attached to it. Once a container with a local translation (or the top-level translation) is reached, the search looks for the identifier within it. If it is not found, then the search continues up the translation tree. The search proceeds only up the tree, never into siblings.

The Style attribute does not have a local translation to it (an attribute can’t). So when it encountered as part of the Audio container, the algorithm looks for a local translation attached to the Audio container and it does find one. Within this container, we find DML ID 1 and we identify the Style attribute as a uint primitive.

Next the parser would encounter the New-Slide container. In this case, our context is the Slideshow container and it has a local translation defined. Within that local translation, DML ID 1 is matched to Slideshow:1 and associated with New-Slide, a container. From within the New-Slide, we encounter a DML ID of 2. We look for a local translation for New-Slide but none is found, so we proceed up the content tree to the Slideshow container. Slideshow has a local translation. There is no DML ID of Slideshow:2, so the DML ID 2 still isn’t matched and now

we proceed up the translation tree. At the global level of the translation we discover that DML ID 2 is associated with “Video” and we have a match.

Within Video, we encounter a DML ID of 1 again. Video has its own local translation, so we look there and begin moving within the translation tree. The DML ID of 1 corresponds to Video:1, matching Style. When we next encounter the Audio DML ID of 3, we are looking in the Video local translation and find no match. We move up the translation tree to the global level and find Audio. Finally, when we encounter Style (ID 1) within the Audio container’s attributes, we identify a local translation associated with Audio. This translation contains ID Audio:1 for Style, now defined as a UInt primitive.

If this is confusing, keep in mind that the recommended practice is to avoid re-using the same name for different types, and we would like to avoid re-using “Style” like this across different local translations. But, software can sometimes end up doing things like this, so we illustrate it.

This completes the primer of Data Markup Language. More detail can be found in the other specification documents.