The Data Access Protocol: DAP Version 4.0

Volume 1: Data Model and Serialized Representation

|  |  |
| --- | --- |
| Date: | May 31, 2012 |
| Revised: | June 26, 2012 |
| Status: | Draft |
|  |  |
| Authors: | John Caron (Unidata) |
|  | Ethan Davis (Unidata) |
|  | David Fulker (Opendap) |
|  | James Gallagher (Opendap) |
|  | Dennis Heimbigner (Unidata) |
|  | Nathan Potter (Opendap) |
|  |  |
| Copyright: | 2012 University Corporation for Atmospheric Research |
|  | and Opendap.org |
|  |  |

Abstract

This document defines the Data Access Protocol (DAP) version 4.0 (referred to also as DAP4). This data transmission protocol is intended to supersede all previous versions of the DAP protocol. DAP4 is designed specifically for science data. The protocol relies on the widely used and stable standards, and is capable of representing a wide variety of scientific data types.

Distribution of this document is unlimited.

This document takes material from the DAP2 specification and the OPULS Wiki page.

|  |  |
| --- | --- |
| Changes: |  |
| 2012.05.24: | Initial Draft |
| 2012.05.27 | Added specification of chunk order |
| 2012.05.28 | Added specification and interpretation of simple queries |
| 2012.05.28 | Added discussion about nested sequences. |
| 2012.05.29 | Formatting changes |
| 2012.6.05 | Removed serialized representation sections and constraint sections until James provides direction. |
| 2012.6.24 | Merge all changes from Gallegher, Potter, and Caron, except as noted. |
| 2012.6.24 | Removed all references to Sequences. |
| 2012.6.24 | Inserted Jame’s version of serialized representation. |
| 2012.6.25 | Added DDX RELAX-NG Grammar. |
| 2012.6.24 | Added (semi-)formal description of the DAP4 serialization scheme. |
| 2012.6.26 | Added: (1) Revised Char type (2) Revised unlimited dimension rules (3) revised MAP rules. (4) Removed HTTP references |

Open Questions as of 6/24/20

1. is Int32 x[\*][\*] legal or not? resolved
2. what is the character encoding for Char? resolved
3. James currently has some element in the MIME header such as byte ordering. Should these be made part of the DataDDX response as a binary header?
4. We need ways to specify at least the following version information
   1. DAP protocol version (i.e. the value 4)
   2. the DDX format version
   3. the peristent data representation version.

Notes on decisions made or that need to be made.

1. Added Char to the list of Atomic types.
2. Added atomic type aliases: Byte=UInt8
3. Variables are distinguished from Fields and map array variables must be Variables.
4. Nested Attributes are not supported.
5. Opaque instances are variable length. If not, then we need to consider adding a Bytestring type.
6. Should we use term Cardinal type versus Atomic type.
7. Enumerations have a basetype that is one of the integer atomic types.
8. Added 2-byte packed representation to XDR.

Contents

[1. Introduction 6](#_Toc328486711)

[2. Requirements 6](#_Toc328486712)

[3. Overall Operation 6](#_Toc328486713)

[4. Characterization of a Data Source 7](#_Toc328486714)

[5. DDX Declarations 7](#_Toc328486715)

[5.1 Non-Data Bearing Declarations versus Data Bearing Declarations 7](#_Toc328486716)

[5.2 Groups 8](#_Toc328486717)

[5.3 Fully Qualified Names 8](#_Toc328486718)

[5.4 Dimensions 9](#_Toc328486719)

[5.5 Enumeration Types 9](#_Toc328486720)

[5.6 Atomic Types 9](#_Toc328486721)

[5.6.1 Integer Types 10](#_Toc328486722)

[5.6.2 Floating-point Types 10](#_Toc328486723)

[5.6.3 String Types 10](#_Toc328486724)

[5.6.4 The Opaque Types 11](#_Toc328486725)

[5.6.5 A Note Regarding Implementation of the Atomic Types 11](#_Toc328486726)

[5.7 Container Types 11](#_Toc328486727)

[5.7.1 The Structure Type. 12](#_Toc328486728)

[5.8 Variables 12](#_Toc328486729)

[5.8.1 Arrays 12](#_Toc328486730)

[5.8.2 Simple Variables 12](#_Toc328486731)

[5.8.3 Structure Variables 13](#_Toc328486732)

[5.8.4 Coverage Variables and Maps 13](#_Toc328486733)

[5.9 Attributes and Arbitrary XML 14](#_Toc328486734)

[5.9.1 Attributes 14](#_Toc328486735)

[5.9.2 Arbitrary XML content 15](#_Toc328486736)

[5.9.3 Attribute and OtherXML Specification and Placement 15](#_Toc328486737)

[5.10 Namespaces 16](#_Toc328486738)

[6. Data Representation 17](#_Toc328486739)

[6.1 Structure of the metadata (DDX) Part 17](#_Toc328486740)

[6.2 Structure of the binary part 17](#_Toc328486741)

[6.3 TheDAP4 Serialized Representation (DSR) 18](#_Toc328486742)

[6.3.1 Order of Serialization 18](#_Toc328486743)

[6.3.2 Variable Representation in the Absence of Variable Dimensions. 19](#_Toc328486744)

[6.3.3 Variable Representation in the Presence of Variable-Length Dimensions. 21](#_Toc328486745)

[6.3.4 CheckSums 21](#_Toc328486746)

[6.3.5 Historical Notes 21](#_Toc328486747)

[6.4 Example responses 21](#_Toc328486748)

[6.4.1 A single scalar 22](#_Toc328486749)

[6.4.2 A single array 22](#_Toc328486750)

[6.4.3 A single structure 23](#_Toc328486751)

[6.4.4 An array of structures 23](#_Toc328486752)

[6.4.5 A single varying array (one varying dimension) 23](#_Toc328486753)

[6.4.6 A single varying array (two varying dimensions) 24](#_Toc328486754)

[6.4.7 A varying array of structures with fields that have varying dimensions 25](#_Toc328486755)

[7. DAP4 Chunked Data Representation. 26](#_Toc328486756)

[7.1 Chunked Format Grammar 27](#_Toc328486757)

[7.2 Lexical Structure 27](#_Toc328486758)

[7.3 Error Chunk Schema 27](#_Toc328486759)

[References 29](#_Toc328486760)

[Appendix A. DAP4 Lexical Elements 30](#_Toc328486761)

[A.1 DDX Lexical Element Syntax 30](#_Toc328486762)

[Appendix B. DAP4 DDX Syntax as a Relax-NG Schema 33](#_Toc328486763)

# Introduction

This specification defines the protocol referred to as the Data Access Protocol, version 4.0 (“DAP4”). In this document ‘DAP’ refers to DAP4 unless otherwise noted.

DAP is intended to be the successor to all previous versions of the DAP (specifically DAP version 2.0). The goal is to provide a very general data model capable of representing a wide variety of existing data sets. The DAP builds upon a number of existing data representation schemes. Specifically, it is influenced by DAP version 2.0[], netCDF-4[], HDF5[], and CDM[].

The DAP is a protocol for access to data organized as variables. It is particularly suited to accesses by a client computer to data stored on remote (server) computers which are networked to the client computer. DAP was designed to hide the implementation of different collections of data. The assumption is that a wide variety of data sets using a wide variety of data schemas can be translated into the DAP protocol for transmission from the server holding that dataset to a client computer for processing.

It is important to stress the discipline neutrality of the DAP and the relationship between this and adoption of the DAP in disciplines other than the Earth sciences. Because the DAP is agnostic as relates to discipline, it can be used across the very broad range of data types encountered in oceanography - biological, chemical, physical and geological. There is nothing that constrains the use of the DAP to the Earth sciences.

# Requirements

The key words “MUST”, “MUST NOT”, “REQUIRED”, “SHALL”, “SHALL NOT”, “SHOULD”, “SHOULD NOT”, “RECOMMENDED”, “MAY” and “OPTIONAL” in this document are to be interpreted as described in RFC 2119 [].

# Overall Operation

The DAP is a stateless protocol that governs clients making requests from servers, and servers issuing responses to those requests. This section provides an overview of the requests and responses (i.e. the messages) which DAP-compliant software MUST support. These messages are used to request information about a server and data made accessible by that server, as well as requesting data values themselves.

The DAP uses two responses to represent a data source. One response, the DDX returns metadata information describing the structure of a request for data. That is, it characterizes the variables, their datatypes, names and attributes. The second response, the DataDDX, returns both the metadata about the request, but also the data that was requested. The DDX and the metadata part of the DataDDX are represented using a specific XML[] representation. The syntax of that representation is defined in Section ?.

The DAP returns error information using an Error response. If a request for any of the three basic responses cannot be completed then an Error response is returned in its place.

The two responses (DDX and DataDDX) are complete in and of themselves so that, for example, the data response can be used by a client without ever requesting either of the two other responses. In many cases, client programs will request the DDX response first before requesting the DataDDX response but there is no requirement they do so and no server SHALL require that behavior on the part of clients.

Operationally, communication between a DAP client and a DAP server uses some underlying already existing protocol. Volume II discusses the appropriate choices for the underlying protocol. Whatever protocol is used, the actual DataDDX response must conform to the Multi-Part MIME encoding described in Section ?.

In addition to these data objects, a DAP server MAY provide additional “services” which clients may find useful. For example, many DAP-compliant servers provide HTML-formatted representations or ASCII representations of a data source’s structure and data. Such additional services are discussed in Volume II of this specification.

# Characterization of a Data Source

The DAP characterizes a data source as a collection of variables, dimensions, and enumeration types. Each variable consists of a name, a type, a value, and a collection of Attributes. Dimensions have a name and a size. Enumerations list names and values of the enumeration constants. These elements may be grouped into collections using the concept of a “group” that has an identifier and defines a naming scope for the elements within it. Groups may contain other groups.

The distinction between information in a variable and in an Attribute is somewhat arbitrary. However, the intention is that Attributes hold information that aids in the interpretation of data held in a variable. Variables, on the other hand, hold the primary content of a data source.

Section ? provides a formal syntax for DAP DDX characterizations. It is defined using the RelaxNG standard [] for describing the context-free syntax of a class of XML documents, the DDX in this case. The following discussion closely follows that RelaxNG syntax specification. It should be noted that any syntax specification requires a specification of the lexical elements of the syntax. The XML specification [] provides most of the lexical context for the syntax, but there are certain places where additional lexical elements must be used. Section ? describes those additional lexical elements, and those elements are discussed at appropriate points in the following discussion.

Since the syntax is context-free, there are semantic limitations on what is legal in a DDX. These semantic limitations are defined at appropriate places in the following documentation. It should also be noted that if there are conflicts between what is described here and the RelaxNG syntax, then the syntax takes precedence.

# DDX Declarations

## Non-Data Bearing Declarations versus Data Bearing Declarations

The declarations in a DDX can be grouped into two classes. One class is non-data bearing. That is, it provides syntactic or structural metadata about a dataset. The non-data bearing declarations are Groups, Dimensions, and Enumerations. Such declarations do not contain data values themselves. In many cases these declarations will not be explicitly represented in the original dataset. Instead, their existence and value(s) will be inferred based on various standards and conventions. The data bearing class of declarations are Variables and Attributes. These elements of the data model are used to house data values or semantic metadata read from the dataset (or, in the latter case) synthesized from the values and standards/conventions that the dataset is known to follow.

## Groups

A group is specified using this XML form.

|  |
| --- |
| <Group name=“name”>  …  </Group> |

A group defines a name space and contains other DAP elements. Specifically, it can contain groups, variables, dimensions, and enumerations. The fact that groups can be nested means that the set of groups in a DDX form a tree structure. For any given DDX, there exists a root group that is the root of this tree.

A nested set of groups defines a variety of name spaces and access to the contents of a group is specified using a notation of the form “/g1/g2/…/gn”. This is called a “path”. By convention “/” refers to the root group. Thus the path “/g1/g2/g3” indicates that one should start in the root group, move to group g1 within that root group, then to group g2 within group g1, and finally to group g3. This is more fully described in the section on Fully Qualified names (Section ?).

For comparison purposes, DAP groups correspond to netCDF-4 groups and not to the more complex HDF5 Group type.

Semantic Limitations

1. If declared, Groups must be named. This includes the root group, but that group the name is ignored for the purposes of fully qualified names.
2. A Group can contain any object, including a Group
3. Each Group declares a new lexical scope for the objects it contains.
4. A Group cannot have dimensions and a Group cannot be defined within a Structure.

## Fully Qualified Names

Every object in a DAP4 Dataset has a Fully Qualified Name (FQN). These names follow the common conventions of lexically-scoped identifiers. To write and FQN for some object O, locate the closest, top-level, enclosing object (P) for O. P may be the same as O. Start by creating the FQN for P by traversing a path through the Group tree to P. Concatenate the group names on that path and separating them with ‘/’. The root group is assumed to have no name, hence the FQN will begin with “/”. The FQN for P will end with the name of P. If O is a field nested in some set of (possibly nested) set of Structures or Enumerations, then collect a field pathname from P to O by concatenating the names on that path and separated by dots (“.”). The last name in the field pathname is the names of P. Prefix the field pathname with “.”. Concatenate the FQN with the field pathname for O to create the final FQN for O.

The forward slash character is never legal as a name. Cases where dots are used in names are accommodated by allowing dots to be escaped using a backslash (\).

## Dimensions

A dimension declaration is specified using this XML form.

|  |
| --- |
| <Dimension name=“name” size=“size”/> |

A dimension declaration will be referenced elsewhere in the DDX by specifying its name. It should also be noted that anonymous dimensions also exist. They have a size but no name. Anonymous dimensions do not need to be declared. Additionally, as discussed in Section ?, an anonymous dimension may be of variable length, and such a variable length dimension is indicated using the notation “\*” for the size.

Semantic Limitations

1. Dimension declarations are not associated with a data type.
2. Dimension sizes MUST be a 64-bit integer or “\*”, which indicates a variable length dimension

## Enumeration Types

An enumeration type defines a set of names with specific values: enumeration constants. As will be seen in Section ?, enumeration types may be used as the type for variables or attributes. The values that can be assigned to such typed objects must come from the set of enumeration constants.

An enumeration type specifies a set of named, integer constants. When a data source has a variable of type 'Enumeration' a DAP 4 server MUST represent that variable using a specified *integer type*, up to an including a 64-bit unsigned integer.

An Enumeration type is declared using this XML form.

|  |
| --- |
| <Enumeration name=“name” basetype=”atomic type”>  <EnumConst name=“name” value=”value”/>  …  </Enumeration> |

Semantic Limitations

1. The optional “basetype” XML attribute defines the type for the value XML attribute of each enumeration constant. This basetype must be one of the integer types (see Section ?). If unspecified, then it defaults to the Atomic type “Int32”.

## Atomic Types

The DAP4 specification assumes the existence of certain pre-defined, declared types called atomic types. As their name suggests, atomic data types are conceptually indivisible. Atomic variables are used to store integers, real numbers, strings and URLs. There are five classes of atomic types, with each family containing one or more variations: integer, floating-point, string, enumerations, and opaque.

### Integer Types

The integer types are summarized in Table 1. The lexical structure for integer constants is defined in Appendix ?.

1. The DAP Integer Data types.

|  |  |  |
| --- | --- | --- |
| Type Name | Description | Range of Legal Values |
| Boolean | Single bit integer | [0, 1] |
| Int8 | Signed 8-bit integer | [-(2^7), (2^7) - 1] |
| UInt8 | Unsigned 8-bit integer | [0, (2^8) - 1] |
| Byte | Synonym for UInt8 |  |
| Int16 | Signed 16-bit integer | [-(2^15), (2^15) - 1] |
| UInt16 | Unsigned 16-bit integer | [0, (2^16) - 1] |
| Int32 | Signed 32-bit integer | [-(2^31), (2^31) - 1] |
| UInt32 | Unsigned 32-bit integer | [0, (2^32) - 1] |
| Int64 | Signed 64-bit integer | [-(2^63), (2^63) - 1] |
| UInt64 | Unsigned 64-bit integer | [0, (2^64) - 1] |

### Floating-point Types

The floating point data types are summarized in Table 2. The two floating point data types use IEEE 754[11] to represent values. The two types correspond to ANSI C’s float and double data types. The lexical structure for floating point constants is defined in Section ?.

1. The DAP Floating-Point Data types.

|  |  |  |
| --- | --- | --- |
| Type Name | Description | Range of Legal Values |
| Float32 | 32-bit Floating-point number | [±1.175494351 × 10^38,  ±3.402823466 × 10^38] |
| Float64 | 64-bit Floating-point number | [±2.2250738585072014 × 10^308, ±1.7976931348623157 × 10^308] |

### String Types

The three string data types are summarized in Table 3. Again, the lexical structure for these is defined in **Error! Reference source not found.**.

Strings are individually sized. This means that in an array of strings, for example, each instance of that string MAY be of a different size.

Note that the Char type is defined to be 7-bit US-ASCII embedded in an 8-bit byte with a zero high order bit. This means that it can represent only a subset of UTF-8.

1. The String Data types.

|  |  |  |
| --- | --- | --- |
| Type Name | Description | Range of Legal Values |
| Char | 8-bit US-ASCII character with the high order bit zero. | [\x00, \x7f] |
| String | A variable length string of UTF-8 characters | As defined in [] |
| URI | A Uniform Resource Identifier | As defined in IETF RFC 2396[3] |

### The Opaque Types

The XML scheme for declaring an Opaque type is as follows.

|  |
| --- |
| <Opaque … > |

The Opaque type is use to hold objects like JPEG images and other Binary Large Object (BLOB) data that have significant internal structure which might be understood by clients (e.g., an image display program) but that would be very cumbersome to describe using DAP's built-in types. Defining a variable of type “Opaque” does not communicate any information about its content, although an attribute could be used to do that.

Semantic Limitations

1. The content of an opaque object is completely un-interpreted by the DAP4 implementation. There is no attempt to re-order four-byte words to or from network byte order and there is no attempt to modify its actual length to conform to, for example, a four-byte boundary, although when transmitted on the wire, padding may be added.
2. The Opaque type is an Atomic Type, which might seem odd because instances of Opaque can be of different sizes. However, by thinking of Opaque as equivalent to a byte-string type, the analog with strings makes it clear that it should be an Atomic type.

### A Note Regarding Implementation of the Atomic Types

When implementing the DAP, it is important to match information in a data source or read from a DAP response to the local data type which best fits those data. In some cases an exact match may not be possible. For example Java lacks unsigned integer types [13]. Implementations faced with such limitations MUST ensure that clients will be able to retrieve the full range of values from the data source. As a practical consideration, this may be implemented by hiding the variable in question or returning an error.

## Container Types

There is currently one container type, namely the Structure type.

### The Structure Type.

A Structure groups a list of fields so that the collection can be manipulated as a single item. A field is syntactically identical to a variable. The Structure’s fields MAY be of any type, including Structure types. The order of items in the Structure is significant only in relation to the serialized representation of that Structure.

## Variables

Each variable in a data source MUST have a name, a type and one or more values. Using just this information and armed with an understanding of the definition of the DAP data types, a program can read any or all of the information from a data source.

The DAP variables come in several different types. There are several atomic types, the basic indivisible types representing integers, floating point numbers and the like, and two constructor types (also called container types) which are flexible collections of other variables. Constructor types may contain both atomic variable types as well as other constructor types.

The DAP variables describe the data when it is being transferred from the server to the client. It does not necessarily describe the format of the data inside the server or client. The DAP defines, for each data type described in this document, a serialized representation, which is the information actually communicated between DAP servers and DAP clients. The serialized representation consists of two parts: the declaration of the type and the serialized encoding of its value(s). The data representation is presented in Section ? and Appendix ?.

### Arrays

Most (but not all) types may be arrays. An Array is a multi-dimensional indexed data structure. An Array’s member variable MUST be of some DAP data type. Array indexes MUST start at zero. Arrays MUST be stored in row-major order (as is the case with ANSI C), which means that the order of declaration of dimensions is significant. The size of each Array’s dimensions MUST be given, except for variable length dimensions. The total number of elements in an Array MUST NOT exceed 2^64-1. There is no prescribed limit on the number of dimensions an Array may have except that the foregoing limit on the total number of elements MUST NOT be exceeded. The number of elements in an Array is fixed as that given by the size(s) of its dimension(s), except when the array has a variable length dimension.

Semantic Limitations

1. Simple variables (see below) may be arrays.
2. Structures MAY be arrays.

### Simple Variables

A simple, dimensioned variable is declared using this XML form.

|  |
| --- |
| <Int32 name=”name”>  <Dimension name=”name”/>  …  <Dimension size=”integer”/>  …  <Dimension size=”\*”/>  </Int32> |

A simple variable is one whose type is one of the Atomic Types (see Section 1). The name of the Atomic Type (Int32 in this example) is used as the XML element name. Within the body of that element, it is possible to specify zero or more dimension references. A dimension reference may refer to a previously defined dimension declaration. It may also define an anonymous dimension with no name, but with a size. It may also define a variable length dimension using a size of “\*”.

Semantic Limitations

1. When declaring a variable, only one variable length dimension may be referenced, and that variable length dimension must be the last dimension listed.
2. Variables must be “top-level”, which means that they are declared immediately within groups. This is to distinguish them from “fields”, which look syntactically like variables, but are declared in Structures.

### Structure Variables

As with simple variables, a structure variable specifies a type as well as any dimension for that variable. The type, however, is a Structure.

#### Structures

The XML scheme for a Structure typed variable is as follows.

|  |
| --- |
| <Structure name=”name”>  {structure body}  …  <Dimension name=”name”/>  …  </Structure> |

The Structure contains within it a “structure body”, which is defined in Section ?. The structure body may be followed with a list of dimension references indicating the dimensions of the Structure typed variable.

Semantic Limitations

1. Structures MAY be dimensioned.

### Coverage Variables and Maps

A Coverage is a concept commonly found in many disciplines, where the term refers to a sampled function with both its domain and range explicitly enumerated by variables. DAP2 uses the name ‘Grid’ to denote what the OGC calls a ‘rectangular grid’ [cite: the abstract coverage spec.]. DAP4 expands on this so that other types of Coverages can be explicitly represented.

In DAP4, the range for a coverage are the values of a (simple or container) variable that includes a specific set of ‘maps’ or ‘coordinate variables’ that define the domain for the sampled function. Taken as whole, this type of variable is called a ‘grid’ for convenience sake.

Using OGC coverage terminology, we have this.

1. The maps specify the ''Domain''
2. The array specifies the ''Range''
3. The Grid is itself a ''Coverage'' per OGC.
4. The Domain and Range are sampled functions

A map is defined using the following XML scheme.

|  |
| --- |
| <Map name=”FQN for some variable defined in the DDX” /> |

An example might look like this.

|  |
| --- |
| <Float32 name=”A”>  <Dimension name=”lat”/>  <Dimension name=”lat”/>  <Map name=”lat”/>  <Map name=”lon”/>  </Float32> |

Where the map variables are defined elsewhere like this.

|  |
| --- |
| <Float32 name=”lat”>  <Dimension name=”lat”/>  </Float32  <Float32 name=”lon”>  <Dimension name=”lon”/>  </Float32> |

The containing variable, A in the example, will be referred to as the “array variable”.

Semantic Limitations

1. Each map variable MUST have a rank no more than that of the array.
2. An array variable can have as many maps as desired.
3. The dimensions of the array variable may not contain duplicates so A[x,x] is disallowed.
4. Any map duplicates are ignored and the order of declaration of the maps is irrelevant.
5. A Map variable may not have a variable length dimension.
6. The fully-qualified name of a map must either be in the same lexical scope as the array variable, or the map must be in some enclosing scope.
7. The set of named “associated dimensions for a map must be a subset of the set of named “associated dimensions” for the array variable.

The term “associated dimensions” is defined as follows.

1. The associated dimensions set is initialized to empty.
2. For each element mentioned in the fully qualified name (FQN) of the map or the array variable, add any named dimensions associated with FQN element to the associated dimensions set (removing duplicates, of course).

In practice, the means that an array variable or map variable must take into account any dimensions associated with any enclosing dimensioned Structure.

## Attributes and Arbitrary XML

### Attributes

Attributes are defined using the following XML scheme.

|  |
| --- |
| <Attribute name=”name” type=”atomic type name”>  <Namespace href=”http://netcdf.ucar.edu/cf”/>  <Value value=”value”/>  …  <Value value=”value”/>  </Attribute> |

In DAP4, Attributes (not to be confused with XML attributes) are tuples with four components:

Name

Type

Vector of values

One or more Namespaces (optional)

This differs slightly from DAP2 Attributes because the *namespace* feature has been added, although clients can choose to ignore it. For more about namespaces, refer to Section ?. The intent of including the namespace information is to simplify interactions with semantic web applications where certain schemas or standards have formal definitions of attributes.

Attributes are typically used to associate semantic metadata with the variables in a data source. Attributes are similar to variables in their range of types and values, except that they are somewhat limited when compared to those for variables.

Attributes defined at the top-level within a group are also referred to as “group attributes”.

While the DAP does not require any particular Attributes, some may be required by various metadata conventions. The semantic metadata for a data source comprises the Attributes associated with that data source and its variables[14]. Thus, Attributes provide a mechanism by which semantic metadata may be represented without prescribing that a data source use a particular semantic metadata convention or standard.

Semantic Limitations

1. DAP4 explicitly treats an attribute with one value as an attribute whose value is a one-element vector.
2. The following types are allowed for Attributes:
3. All of the Atomic types are allowed as the type for an attribute
4. String typed Attributes use UTF-8 encoding and Char typed attributes use US-ASCII encoding.
5. Attribute value constants MUST conform to the appropriate constant format for the given attribute type and as defined in Appendix ?.

### Arbitrary XML content

By supporting an explicit type to hold “arbitrary XML” markup, DAP4 provides a way for the protocol to transport information encoded in XML along with the attributes read from the dataset itself. This has proved very useful in work with semantic web software.

In an XML representation of DAP4, the name is optional, the XML element is *<OtherXML/>* and there are no *<value/>* elements because the “other xml” appears as the content of the *<OtherXML/>* element. The value of the attribute must be valid XML and must be distinct from the XML markup used to encode elements of the DAP4 data model (i.e., in a practical sense, the content of an <OtherXML> attribute will be in a namespace other than DAP4).

### Attribute and OtherXML Specification and Placement

Attribute and OtherXML declarations MAY occur within the body of the following XML elements: Group, Dimension, Variable, Field, and Structure.

## Namespaces

All elements of the DDX, Groups, Dimensions, Variables, and Attributes can contain an associated Namespace element. The namespace’s value is defined in the form of an XML style URI string defining the context for interpreting the element containing the namespace. Suppose, hypothetically, that we wanted to specify that an Attribute is to be interpreted as a CF convention []. One might specify this as follows.

|  |
| --- |
| <Attribute name=”latitude”>  <Namespace href=”http://cf.netcdf.unidata.ucar.edu”/>  …  </Attribute> |

Note that this is not to claim that this is how to specify a CF convention; this is purely hypothetical.

# Data Representation

Data can be an elusive concept. Data may exist in some storage format on some disk somewhere, on paper somewhere else, in active memory on some server, or transmitted along some wire between two computers. All these can still represent the same data. That is, there is an important distinction to be made between the data and its representation. The data consist of numbers: abstract entities that usually represent measurements of something, somewhere. Data also consist of the relationships between those numbers, as when one number defines a time at which some quantity was measured.

The abstract existence of data is in contrast to its concrete representation, which is how we manipulate and store it. Data can be stored as ASCII strings in a file on a disk, or as twos-complement integers in the memory of some computer, or as numbers printed on a page. It can be stored in netCDF, HDF5, GRIB, JGOFS, a relational database and any number of other digital storage forms.

The DAP specifies a particular representation of data, to be used in transmitting that data from one computer to another. This representation of some data is sometimes referred to as the serialized representation of that data, as distinguished from the representations used in some computer’s memory. The DAP standard outlined in this document has nothing at all to say about how data is stored or represented on either the sending or the receiving computer. The DAP transmission format is completely independent of these details.

## Structure of the metadata (DDX) Part

The response document, called the DataDDX, uses the Multipart-Mime standard [cite: Multipurpose Internet Mail Extensions (MIME) Part One: Format of Internet Message Bodies]. The response is the server's answer to a request for data from a client. A response will consist of two parts (in the Multipart-MIME sense):

1. A DDX that has no attribute information and contains (only) the variables requested; and
2. A binary part that contains the data for those variables

The first part always contains the DDX. Note that the Content-Type of this part is “text/xml” and that its charset parameter is UTF-8. Note also that the transfer encoding is binary. The boundary is a unique 128 bit value used to separate the Multipart-MIME parts.

|  |
| --- |
| --<<boundary>> |
| Content-Type: text/xml; charset=UTF-8 |
| Content-Transfer-Encoding: binary |
| Content-Description: ddx |
| Content-Id: <<start-id>> |
|  |
| <<DDX here>> |
| --<<boundary>> |
| ... |

## Structure of the binary part

The binary part starts with the MIME headers for a Part in a multipart-related document. entire binary content of the response is contained in a second Multipart-MIME part. Note that the “Content-Type” of this part is “application/x-dap-big-endian” or “application/x-dap-little-endian”. The client will use this header to correctly decode data values. The “Content-Length” header is present here to help internet tools (such as caches) when the server can realistically know the size of the data to be serialized before the serialization takes place. A value of -1 indicates an unknown size.

|  |
| --- |
| ... |
| --<<boundary>> |
| Content-Type: application/x-dap-little-endian |
| Content-Transfer-Encoding: binary |
| Content-Description: data |
| Content-Id: <<next-id>> |
| Content-Length: <<-1 or the size in bytes of the binary data>> |
|  |
| <<serialized data>> |
| --<<boundary>> |

The serialized data is actually encoded using two, layered encodings. The first (“top”) encoding serializes all of the response data into a single long sequence of bytes. This encoding is described in the following Section (Section ?). The second (“bottom”) encoding treats the top level encoding as a sequence of un-interpreted bytes that is broken into varying sized “chunks”. Thus the receiver of a response obtains these chunks in order. By (at least conceptually) concatenating all of these chunks in order and with the chunk header removed, the complete serialized data response can be obtained. The specification of this chunked encoding is described in Section ?.

## TheDAP4 Serialized Representation (DSR)

Given a DDX and the corresponding data, the serialized representation is formally described in this section.

A note on dimension ordering. Consider this example.

|  |
| --- |
| <Int32 name=”i”> |
| <Dimension name=”d1”/> |
| <Dimension name=”d2”/> |
| … |
| <Dimension name=”dn”/> |
| </Int32> |

The dimensions are considered ordered in the left-to-right order [d1][d2]…[dn]. The assumption of row-major order means that in enumerating all possible combinations of these dimensions, the right-most is considered to vary the fastest. The terms “right(most)” or “left(most”) refer to this ordering of dimensions.

### Order of Serialization

The data appearing in a serialized representation is the concatenation of the variables specified in the tree of Groups within a DDX, where the variables in a group are taken in depth-first, top-to-bottom order. The term “top-to-bottom” refers to the lexical ordering of the variables in an XML document specifying a given DDX.

If a variable is a Structure variable, then its data representation will be the concatenation of the variables it contains, which will appear in top-to-bottom order.

If a variable or field has dimensions, then the contents of each dimensioned data item will appear concatenated and taken in row-major order.

### Variable Representation in the Absence of Variable Dimensions.

Given a dimensioned variable, with no dimension being variable length, it is represented as a 64-bit integer count, N say (giving the number of elements) followed by the N scalar values listed in row-major order. The count is computed by taking the product of the sizes of all the dimensions. So, for example, A[2][3][4] has an element count of 2x3x4 = 24.

If the variable is scalar, then it is represented as a single scalar value with no preceding count.

#### Numeric Scalar Atomic Types

For the numeric atomic types, scalar instances are represented as follows. In all cases a consistent byte-ordering is assumed, but the choice of byte order is at the discretion of the program that generates the serial representation, typically a server program.

|  |  |  |
| --- | --- | --- |
| Type Name | Description | Representation |
| Int8 | Signed 8-bit integer | 8 bits |
| UInt8 | Unsigned 8-bit integer | 8 bits |
| Int16 | Signed 16-bit integer | 16 bits |
| UInt16 | Unsigned 16-bit integer | 16 bits |
| Int32 | Signed 32-bit integer | 32-bits |
| UInt32 | Unsigned 32-bit integer | 32-bits |
| Int64 | Signed 64-bit integer | 64-bits |
| UInt64 | Unsigned 64-bit integer | 64-bits |
| Float32 | 32-bit IEEE floating point | 32-bits |
| Float64 | 64-bit IEEE floating point | 64-bits |
| Char | Unsigned 8-bit integer | Treat same as UInt8 |

In narrative form: all numeric quantities are used as a raw, unsigned vector of N bytes, where N is 1 for Char, Int8, and UInt8; it is 2 for Int16 and UInt16; it is 4 for Int32, UInt32, and Float32; and it is 8 for Int64, UInt64, and Float64. If the server chooses to byte swap the values, then the following swapping rules are used. Not that a size of 16 is also included. This is used solely to represent checksums as 128-bit unsigned integers (see Section ?).

|  |  |
| --- | --- |
| Size (in bytes) | Byte Swapping Rules |
| 1 | Not Applicable. |
| 2 | Byte 0 -> Byte 1  Byte 1 ->Byte 0 |
| 4 | Byte 0 -> Byte 3  Byte 1 ->Byte 2  Byte 2 -> Byte 1  Byte 3 ->Byte 0 |
| 8 | Byte 0 -> Byte 7  Byte 1 ->Byte 6  Byte 2 -> Byte 5  Byte 3 ->Byte 4  Byte 4 -> Byte 3  Byte 5 ->Byte 2  Byte 6 -> Byte 1  Byte 7 ->Byte 0 |
| 16 | Byte 0 -> Byte 15  Byte 1 ->Byte 14  Byte 2 -> Byte 13  Byte 3 ->Byte 12  Byte 4 -> Byte 11  Byte 5 ->Byte 10  Byte 6 -> Byte 9  Byte 7 ->Byte 8  Byte 8 -> Byte 7  Byte 9 ->Byte 6  Byte 10 -> Byte 5  Byte 11 ->Byte 4  Byte 12 -> Byte 3  Byte 13 ->Byte 2  Byte 14 -> Byte 1  Byte 15 ->Byte 0 |

#### Variable-Length Scalar Atomic Types

|  |  |  |
| --- | --- | --- |
| Type Name | Description | Representation |
| String | Vector of 8-bit bytes representing a UTF-8 String | The number of bytes in the string (in UInt64 format) followed by the bytes. |
| URL | Vector of 8-bit bytes representing a URL | Same as String |
| Opaque | Vector of un-interpreted 8-bit bytes | The number of bytes in the vector (in UInt64 format) followed by the bytes. |

In narrative form, instances of String, Opaque, and URL types are represented as a 64 bit length (treated as UInt64) of the instance followed by the vector of bytes comprising the value.

#### Structure Variable Representation

A Structure typed variable is represented as the concatenation of the representations of the variables contained in the Structure taken in top-to-bottom order. This representation may be recursive if one of the fields itself is a Structure variable. Dimensioned structures are represented in a form analogous to dimensioned variables of atomic type. The Structure array is preceded by a count of the number of instances and followed by that many instances of the Structure, where the instances are listed in row-major order.

### Variable Representation in the Presence of Variable-Length Dimensions.

Given a dimensioned variable, with at least one dimension being variable length, it is represented as follows.

First, it is preceded by a 64-bit integer count (giving the number of elements). The count is the product up to the first variable length dimension. For example, an array of the form A[2][3][\*][7][\*] has an element count of 2x3 = 6 and all dimensions to the right and including the leftmost varying length dimension are ignored.

Second, instead of the count being valued by the actual values, the count, N say, is followed by N “variable-length vectors”. Each such vector consists of a length, L say, in UInt64 form and giving the number of elements for a specific occurrence of the variable-length dimension. So in our example, there will be 6 variable-length vectors concatenated together. Note that the length, L, of each of the variable length vectors may differ. Section ? provides some examples in detail.

This representation is recursive in that the values in a variable-length vector may themselves be variable-length vectors (as in our example above). Each such vector is encoding using the rules above. In our example, then, there will be 6 variable-length vectors, each of which in turn consists of 7 nested varying-length vectors.

### CheckSums

Checksums will be computed for the values of all the variables at the top-level of each Group in the response. The checksum value will follow the value of the variable. The checksum algorithm defaults to MD5 (chosen primarily for performance reasons.

Checksum values will be written as 128-bit values using the endian-ness specified for the serialized form.

### Historical Notes

The encoding described in Section ? is essentially the same as the serialization form of the DAP2 protocol [cite], but has been extended to support arrays with varying dimensions, had Sequences removed, and stripped of redundant information added by various XDR implementations.

The DAP4 Serialization rules are derived from, but not the same as, XDR [cite]. The differences are as follows.

1. Values are encoded using the byte order of the server. This is the so-called “receiver makes it right” rule.
2. No padding is used.
3. Floating point values always use the IEEE 754 standard.
4. One and two-byte values are not be converted to four byte values.

## Example responses

In these examples, spaces and newlines have been added to make them easier to read. The real responses are as compact as they can be. Since this proposal is just about the form of the response - and it really focuses on the BLOB part - there no mention of 'chunking.' For information on how this BLOB will/could be chunked, see Section ?. NB: Some poetic license used in the following and the checksums for single integer values seems silly, but these are really simple examples.

### A single scalar

|  |
| --- |
| ... |
| Content-Type:multipart/related; type="text/xml"; start="<<start id>>"; boundary="<<boundary>>" |
|  |
| --<<boundary>> |
| Content-Type: text/xml; charset=UTF-8 |
| Content-Transfer-Encoding: binary |
| Content-Description: ddx |
| Content-Id: <<start-id>> |
|  |
| <Group name=”foo”> |
| <Int32 name=”x”/> |
| </Group> |
| --<<boundary>> |
| Content-Type: application/x-dap-little-endian |
| Content-Transfer-Encoding: binary |
| Content-Description: data |
| Content-Id: <<next-id>> |
| Content-Length: <<-1 or the size in bytes of the binary data>> |
|  |
| x |
| <<checksum>> |
|  |
| --<<boundary>> |

### A single array

|  |
| --- |
| <Group name=”foo”> |
| <Int32 name=”x”? |
| <Dimension size=”2”/> |
| <Dimension size=”4”/> |
| </Int32> |
| </Group> |
|  |
| ... |
| Content-Length: <<-1 or the size in bytes of the binary data>> |
|  |
| x00 x01 x02 x03 x10 x11 x12 x13 |
| <<checksum>> |
|  |
| --<<boundary>> |

### A single structure

Note that in this example, there is a single variable at the top-level of the root Group / and that is s, so it is s that we compute the checksum for.

### An array of structures

|  |
| --- |
| <Group name=”foo”> |
| <Structure name=”s”> |
| <Int32 name=”x”> |
| <Dimension size=”2”/> |
| <Dimension size=”4”/> |
| </Int32> |
| <Float64 name=”y”/> |
| <Dimension size=”3”/> |
| </Structure> |
| </Group> |
|  |
| ... |
| Content-Length: <<-1 or the size in bytes of the binary data>> |
|  |
| x00 x01 x02 x03 x10 x11 x12 x13 |
| y |
| x00 x01 x02 x03 x10 x11 x12 x13 |
| y |
| x00 x01 x02 x03 x10 x11 x12 x13 |
| y |
| <<checksum>> |
|  |
| --<<boundary>> |

### A single varying array (one varying dimension)

|  |
| --- |
| <Group name=”foo”> |
| <String name=”s”/> |
| <Int32 name=”a”> |
| <Dimension size=”\*”/> |
| </Int32> |
| <Int32 name=”x”> |
| <Dimension size=”2”/> |
| <Dimension size=”\*”/> |
| </Int32> |
| </Group> |
| ... |
| Content-Length: <<-1 or the size in bytes of the binary data>> |
|  |
| 16 This is a string |
| <<checksum>> |
|  |
| 5 a0 a1 a2 a3 a4 |
| <<checksum>> |
|  |
| 3 x00 x01 x02 6 x00 x01 x02 x03 x04 x05 |
| <<checksum>> |
|  |
| --<<boundary>> |

Notes:

1. The checksum calculation includes only the values of the variable, not the prefix length bytes.
2. The varying dimensions are treated 'like strings' and prefixed with a length count. In the last of the three variables, the array x is a 2 by varying array with the example's first 'row' containing 3 elements and the second 6.

### A single varying array (two varying dimensions)

|  |
| --- |
| <Group name=”foo”> |
| <Int32 name=”x”> |
| <Dimension size=”\*”/> |
| <Dimension size=”\*”/> |
| </Int32> |
| </Group> |
| ... |
| Content-Length: <<-1 or the size in bytes of the binary data>> |
|  |
| 3 |
|  |
| 3 x00 x01 x02 |
|  |
| 6 x10 x11 x12 x3 x14 x15 |
|  |
| 1 x20 |
| <<checksum>> |
|  |
| --<<boundary>> |
|  |
| A varying array of structures |
| <Group name=”foo”> |
| <Structure name=”s”> |
| <Int32 name=”x”> |
| <Dimension size=”4”/> |
| <Dimension size=”4”/> |
| </Int32> |
| <Float64 name=”y”/> |
| <Dimension size=”\*”/> |
| </Structure> |
| </Group> |
| ... |
| Content-Length: <<-1 or the size in bytes of the binary data>> |
|  |
| 2 |
|  |
| x00 x01 x02 x03 x10 x11 x12 x13 |
| y |
|  |
| x00 x01 x02 x03 x10 x11 x12 x13 |
| y |
| <<checksum>> |
|  |
| --<<boundary>> |

Note that two rows are assumed.

[Caron: I would recommend some kind of an <end> tag, rather than having to know the number of structures that will get returned before you start writing.]

### A varying array of structures with fields that have varying dimensions

|  |
| --- |
| <Group name=”foo”> |
| <Structure name=”s”> |
| <Int32 name=”x”> |
| <Dimension size=”2”/> |
| <Dimension size=”\*”/> |
| </Int32> |
| <Float64 name=”y”/> |
| <Dimension size=”\*”/> |
| </Structure> |
| </Group> |
| ... |
| Content-Length: <<-1 or the size in bytes of the binary data>> |
|  |
| 3 |
|  |
| 1 x00 4 x10 x11 x12 x13 |
| y |
|  |
| 3 x00 x01 x02 2 x10 x11 |
| y |
|  |
| 2 x00 x01 2 x10 x11 |
| y |
| <<checksum>> |
|  |
| --<<boundary>> |

# DAP4 Chunked Data Representation.

An important capability for DAP4 is supporting client in determining when a data transmission fails. In order to support such a capability, the DAP4 protocol uses a simplified variation on the HTTP/1.1 chunked transmission format [cite] to serialize the data Part of the response document so that errors are simple to detect. Furthermore, this format is independent of the form or content of that part of the response, so the same format can be used with different response forms or dropped when/if DAP is used with protocols that support out-of-band error signaling, simplifying our ongoing refinement of the protocol.

The data part of a response document is'chunked' in a fashion similar to that outlined in HTTP/1.1. However, in addition to a prefix indicating the size of the chunk, DAP4 includes a chunk-type code. This provides a way for the receiver to know if the next chunk is part of the data response or if it contains an error response (Section ?). In the latter case, the client should assume that the data response has ended, even though the correct closing information was not provided.

Each chunk is prefixed by a chunk header consisting of a chunk type and byte count, all contained in a single four-byte word and encoded using network byte order. The chunk type will be encoded in the high-order byte of the four-byte word and chunk size will be given by the three remaining bytes of that word. The maximum chunk size possible is 2^24 (16 777 216) bytes. Immediately following the four-byte chunk header will be chunk-count bytes followed by another chunk header. More precisely the initial four bytes of the chunk are decoded using the following steps.

1. Treat the 32 bit header a single unsigned integer.
2. Convert the integer from network byte order to the local machine byte order by swapping bytes as necessary (see Section ?). Let the resulting integer be called H.
3. Compute the chunk type by the following expression: type = (H >> 24) & 0xff (Using C-language operators).
4. Compute the chunk length by the following expression: length = (H & 0x00ffffff) (Using C-language operators).

Three chunk-type types are defined in this proposal:

Data

This chunk header prefixes the next chunk in the current data response

Error

This chunk header prefixes an error message; the current data response has ended

End

This chunk header is the last one for the current data response

[JohnCaron:

(1)Perhaps "message" is a better name than "chunk' ?

2) I would not limit the size of the chunk to 2^24. In fact, I would use a base-128 variable length encoding to let the size be as large as needed, without wasting space.]

## Chunked Format Grammar

|  |
| --- |
| chunked\_response: chunklist ; |
|  |
| chunklist: chunk | chunklist chunk; |
|  |
| /\* Semantic limitation: the number of bytes in the CHUNKDATA must be equal to SIZE\*/ |
| chunk: CHUNKTYPE SIZE CHUNKDATA; |

## Lexical Structure

|  |
| --- |
| /\* A single 8-bit byte, with the encoding 0 = data, 1 = error, 2 = end \*/ |
| CHUNKTYPE = '\x01'|'\x02'|'\0x03' |
|  |
| /\* A sequence of three 8-bit bytes, interpreted as an integer on network byte order \*/ |
| SIZE = [\0x00-\0xFF][\0x00-\0xFF][\0x00-\0xFF] |
|  |
| CHUNKDATA = [\0x00-\0xFF]\* |

## Error Chunk Schema

The error chunk is defined to be an XML document as defined by the following RELAX-NG schema.

|  |
| --- |
| <grammar xmlns="http://relaxng.org/ns/structure/1.0" |
| xmlns:doc="http://www.example.com/annotation" |
| datatypeLibrary="http://www.w3.org/2001/XMLSchema-datatypes" |
| ns="http://xml.opendap.org/ns/DAP/4.0#" |
| > |
| <start> |
| <ref name="error"/> |
| </start> |
| <define name="error"> |
| <element name="Error"> |
| <attribute name="errorcode"><data type="integer"/></attribute> |
| <element name = "Message"><text/></Message> |
| <optional> |
| <interleave> |
| <element name = "Position"><text/></Message> |
| <element name = "Context"><text/></Message> |
| <element name = "OtherInformation"><text/></Message> |
| </interleave> |
| </optional> |
| </element> |
| </define> |

References

1. DAP4 Lexical Elements
   1. DDX Lexical Element Syntax

This section describes the lexical elements that occur in the DAP4 DDX.

Within the RelaxNG DAP4 grammar (Section ?) there are markers for occurrences of primitive type such as integers, floats, or strings (ignoring case).

|  |
| --- |
| <attribute name="namespace">  <datatype="string"/>  </attribute> |

The markers typically look like this when defining an attribute that can occur in the DAP4 DDX.

The "<data type="string"/>" specifies the lexical class for the values that this attribute can have. In this case, the namespace attribute is defined to have a string value. Similar notation is used for values occurring as text within an xml element.

The lexical specification later in this section defines the legal lexical structure for such lexical items. Specifically, it defines the format of the following lexical items.

1. Constants, namely: string, float, integer, character, and opaque.
2. Identifiers
3. Fully qualified names (also referred to as FQN) (see Section ?).

The specification is written using the extended Posix regular expression notation [] with some additions.

1. Names are assigned to regular expressions using the notation “name = <regularexpression>”
2. Named expressions can be used in subsequent regular expressions by using the notation “{name}”. Such occurrences are equivalent to textually substituting the expression associated with name for the “{name}” occurrence. This is similar to the way a macro operates.

Note that a regular expression name must be defined before any use to avoid circular definitions.

Notes:

1. The definition of {UTF8} is deferred to the next section.
2. Comments are indicated using the "//" notation. Standard xml escape formats (&x#DDD; or &<name>;) are assumed to be allowed anywhere.
   * 1. Basic character set definitions

|  |
| --- |
| CONTROLS = [\x00-\x1F] // ASCII control characters  WHITESPACE = [ \r\n\t\f]+  HEXCHAR = [0-9a-zA-Z]  // ASCII printable characters  ASCII = [0-9a-zA-Z !"#$%&'()\*+,-./:;<=>?@[\\\]\\^\_`|{}~] |

* + 1. Ascii characters that may appear unescaped in Identifiers

This is assumed to be basically all ASCII printable characters except these characters: '.' '/' '"' ''' and '&'. Occurrences of these characters are assumed to be representable using the standard xml &<name>; notation (e.g. &amp;). In this expression, backslash is interpreted as an escape character.

|  |
| --- |
| IDASCII=[0 9a zA Z!#$%()\*+:;<=>?@\[\]\\^\_`|{}~] |

* + 1. The Numeric Constant Classes: integer and float

|  |
| --- |
| INTEGER = {INT}|{UINT}|{HEXINT}  INT = [+-][0-9][0-9]\*{INTTYPE}?  UINT = [0-9][0-9]\*{INTTYPE}?  HEXINT = {HEXSTRING}{INTTYPE}?  INTTYPE = ([BbSsLl]|"ll"|"LL")  HEXSTRING = (0[xX]{HEXCHAR}{HEXCHAR}\*)  FLOAT = ({MANTISSA}{EXPONENT}?)|{NANINF}  EXPONENT = ([eE][+-]?[0-9]+)  MANTISSA = [+-]?[0-9]\*\.[0-9]\*  NANINF = (-?inf|nan|NaN) |

* + 1. The String Constant Class

|  |
| --- |
| STRING = ([^"\\&]|{XMLESCAPE})\*  CHAR = ([^'\\&]|{XMLESCAPE}) |

* + 1. The Opaque Constant Class

|  |
| --- |
| OPAQUE = 0x([0-9A-Fa-f] [0-9A-Fa-f])+ |

* + 1. The Identifier Class

|  |
| --- |
| ID = {IDCHAR}{IDCHAR}\*  IDCHAR = ({IDASCII}|{XMLESCAPE}|{UTF8})  XMLESCAPE = [&][#][0-9]+; |

* + 1. The Atomic Type Class

|  |
| --- |
| ATOMICTYPE = Char | Byte  | Int8 | UInt8 | Int16 | UInt16  | Int32 | UInt32 | Int64 | UInt64  | Float32 | Float64  | String | URL  | Enumeration | Opaque |

This list should be consistent with the atomic types in the grammar.

* + 1. The Fully Qualified Name Class

|  |
| --- |
| FQN = ([/]{ID})+([.]{ID})\* |

This should be consistent with the definition in Section ?.

* + 1. Lexical Class Precedence

Note that the above lexical element classes are not disjoint. The type element “<datatype=…/>” should be sufficient to interpret the type within the DDX.

* + 1. UTF-8

The UTF-8 specification, <http://www.w3.org/2005/03/23-lex-U>, defines several ways to validate a UTF-8 string of characters.

The full (most correct) validating version of UTF8 character set is as follows.

|  |
| --- |
| UTF8 = ([\xC2-\xDF][\x80-\xBF])  | (\xE0[\xA0-\xBF][\x80-\xBF])  | ([\xE1-\xEC][\x80-\xBF][\x80-\xBF])  | (\xED[\x80-\x9F][\x80-\xBF])  | ([\xEE-\xEF][\x80-\xBF][\x80-\xBF])  | (\xF0[\x90-\xBF][\x80-\xBF][\x80-\xBF])  | ([\xF1-\xF3][\x80-\xBF][\x80-\xBF][\x80-\xBF])  | (\xF4[\x80-\x8F][\x80-\xBF][\x80-\xBF]) |

The lines of the above expression cover the UTF-8 characters as follows:

1. non-overlong 2-byte
2. excluding overlongs
3. straight 3-byte
4. excluding surrogates
5. straight 3-byte
6. planes 1-3
7. planes 4-15
8. plane 16

Note that ASCII and control characters are not included.

The above reference also defines some alternative regular expressions.

There is what is termed the partially-relaxed version of UTF8 defined by this regular expression.

|  |
| --- |
| UTF8 = ([\xC0-\xD6][\x80-\xBF])  | ([\xE0-\xEF][\x80-\xBF][\x80-\xBF])  | ([\xF0-\xF7][\x80-\xBF][\x80-\xBF][\x80-\xBF]) |

Second, there is what is termed the most-relaxed version of UTF8 defined by this regular expression.

|  |
| --- |
| UTF8 = ([\xC0-\xD6]...)|([\xE0-\xEF)...)|([\xF0 \xF7]...) |

Any conforming DAP4 implementation MUST use at least the most-relaxed expression for validating UTF-8 character strings, but MAY use either the partially-relaxed for full validatation expression.

1. DAP4 DDX Syntax as a Relax-NG Schema

<!-- Relax NG Grammar -->

<grammar xmlns="http://relaxng.org/ns/structure/1.0"

xmlns:doc="http://www.example.com/annotation"

datatypeLibrary="http://www.w3.org/2001/XMLSchema-datatypes"

ns="http://xml.opendap.org/ns/DAP/4.0#"

>

<start>

<ref name="rootgroup"/>

</start>

<define name="dataset">

<element name="Group">

<optional>

<attribute name="dapVersion">

<data type="string"/></attribute>

<attribute name="ns"><data type="string"/></attribute>

<attribute name="base"

ns="http://www.w3.org/XML/1998/namespace">

<text/>

</attribute>

</optional>

<ref name="groupdef"/>

</element>

</define>

<define name="groupdef">

<attribute name="name"><data type="ID"/></attribute>

<ref name="metadata"/>

<zeroOrMore>

<interleave>

<ref name="enumdef"/>

<ref name="dimdef"/>

<ref name="variable"/>

</interleave>

</zeroOrMore>

<zeroOrMore>

<ref name="groupdef"/>

</zeroOrMore>

</define>

<define name="enumdef">

<element name="Enumeration">

<attribute name="name"><data type="ID"/></attribute>

<attribute name="basetype">

<data type="ATOMICTYPE"/>

</attribute>

<oneOrMore><ref name="enumconst"/></oneOrMore>

</element>

</define>

<define name="enumconst">

<element name="elt">

<attribute name="name"><data type="ID"/></attribute>

<attribute name="value"><data type="integer"/></attribute>

</element>

</define>

<define name="namespacedef">

<zeroOrMore>

<element name="Namespace">

<attribute name="href"><data type="string"/></attribute>

</element>

</zeroOrMore>

</define>

<define name="dimdef">

<element name="Dimension">

<attribute name="name"><data type="ID"/></attribute>

<attribute name="size">

<data type="integer"/>

</attribute>

<ref name="metadata"/>

</element>

</define>

<define name="dimref">

<element name="Dimension">

<attribute name="name"><data type="FQN"/></attribute>

<attribute name="size"><data type="integer"/></attribute>

</element>

</define>

<define name="vardimref">

<!-- only value allowed is \*-->

<attribute name="size"><data type="string"/></attribute>

</define>

<define name="dimset">

<optional>

<oneOrMore><ref name="dimref"/></oneOrMore>

<optional><ref="vardimref"/></optional>

</optional>

</define>

<define name="atomictype">

<!-- This must be consistent with "variable" below -->

<choice>

<value>Bit</value>

<value>Char</value>

<value>Byte</value>

<value>UByte</value>

<value>Int16</value>

<value>UInt16</value>

<value>Int32</value>

<value>UInt32</value>

<value>Int64</value>

<value>UInt64</value>

<value>Float32</value>

<value>Float64</value>

<value>String</value>

<value>Enumeration</value>

<value>Opaque</value>

</choice>

</define>

<define name="enumref">

<attribute name="name"><data type="FQN"/></attribute>

<attribute name="value"><data type="ID"/></attribute>

</define>

<define name="variable">

<choice>

<ref name="simplevariable"/>

<ref name="structurevariable"/>

</choice>

</define>

<define name="simplevariable">

<choice>

<!-- Following must be consistent with "atomictype" -->

<element name="Char" ><ref name="variabledef"/></element>

<element name="Byte" ><ref name="variabledef"/></element>

<element name="UByte" ><ref name="variabledef"/></element>

<element name="Int16" ><ref name="variabledef"/></element>

<element name="UInt16" ><ref name="variabledef"/></element>

<element name="Int32" ><ref name="variabledef"/></element>

<element name="UInt32" ><ref name="variabledef"/></element>

<element name="Int64" ><ref name="variabledef"/></element>

<element name="UInt64" ><ref name="variabledef"/></element>

<element name="Float32"><ref name="variabledef"/></element>

<element name="Float64"><ref name="variabledef"/></element>

<element name="String" ><ref name="variabledef"/></element>

<element name="Enumeration">

<ref name="enumref"/>

<ref name="variabledef"/>

</element>

<element name="Opaque">

<attribute name="size"><data type="integer"/></attribute>

<ref name="variabledef"/>

</element>

</choice>

</define>

<define name="variabledef">

<attribute name="name"><data type="ID"/></attribute>

<ref name="dimset"/>

<ref name="mapvars"/>

<ref name="metadata"/>

</define>

<define name="mapvars">

<zeroOrMore>

<ref name="mapref"/>

</zeroOrMore>

</define>

<define name="mapref">

<element name="Map">

<attribute name="name"><data type="FQN"/></attribute>

</element>

</define>

<define name="structurevariable">

<element name="Structure">

<attribute name="name"><data type="ID"/></attribute>

<ref name="dimset"/>

<zeroOrMore>

<ref name="variable"/>

</zeroOrMore>

<ref name="metadata"/>

</element>

</define>

<define name="metadata">

<interleave>

<zeroOrMore>

<element name="OtherXML"><ref name="otherxml"/></element>

<element name="Attribute">

<refname="attributedef"/>

</element>

<element name="Namespace"><ref name="string"/></element>

</zeroOrMore>

</define>

<define name="attributedef">

<attribute name="name"><data type="ID"/></attribute>

<optional>

<attribute name="type">

<ref name="atomictype"/>

</attribute>

</optional>

<zeroOrMore>

<optional>

<element name="Namespace">

<attribute name="name"><data type="string"/></attribute>

</element>

</optional>

<element name="Value">

<attribute name="value"><data type="string"/></attribute>

</element>

</zeroOrMore>

</define>

<define name="otherxml">

<element>

<anyName/>

<zeroOrMore>

<choice>

<attribute>

<anyName/>

</attribute>

<text/>

<ref name="otherxml"/>

</choice>

</zeroOrMore>

</element>

</define>

</grammar>