Martin Westhead, EPCC, University of Edinburgh
et. al….

10th September 2003

Data Format Description Language – Primitive Type Ontology

Status of This Memo

This memo provides information to the Grid community regarding the specification of a Data Format Description Language. The specification is currently an early draft which does not represent a consensus within the group. Distribution is unlimited.

Abstract

XML provides an essential mechanism for transferring data between services in an application and platform neutral format. However it is not well suited to large datasets with repetitive structures, such as large arrays or tables. Furthermore, many legacy systems and valuable data sets exist that do not use the XML format. The aim of this working group is to define an XML-based language, the Data Format Description Language (DFDL), for describing the structure of binary and character encoded (ASCII/Unicode) files and data streams so that their format, structure, and metadata can be exposed. This effort specifically does not aim to create a generic data representation language. Rather, DFDL endeavors to describe existing formats in an actionable manner that makes the data in its current format accessible through generic mechanisms.

This document defines the ontology of primitive types.

Contents

1. Purpose of this Ontology

This document provides the first of two basic ontologies. This is the ontology of primitive types. The definition of these types is kept separate from the definition of the Structural Description Language (SDL) since we envisage some groups wanting to provide alternative primitive type definitions, such as SQL types or XML Schema types.

2. XML Schema additions

The schema modifications are as follows:

1. Addition of an attribute group "primitive attributes" which contains attributes common to the primitive types (currently contains just "byteOrder")

Need to think about what attributes to have on these

2. Extension of the typeOfType to typeOfPrim that includes the new attribute group
3. The following elements were added:
    - byte
    - short
    - int
    - long
    - char
    - float
    - double
    - boolean
    - digit
    - letter
    - alphanumeric
    - nonAlphanumeric
    - linefeed
    - carriageReturn
    - tab
    - whiteSpace
    - comma
    - fullStop
    - null
    - minusSign

This list is almost certainly NOT the right one what are we missing? what should we exclude? What are the principles for making these decisions? (Use cases?)

I have (without any strong intention) defined char as a C-like 8-bit thing. We need to worry about Unicode characters and 16-bit Java characters.

The schema was given the namespace "http://www.dfdl.org/2003/primitives"

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- edited with XML Spy v4.4 U (http://www.xmlspy.com) by Martin Westhead (EPCC) -->
<xs:schema targetNamespace="http://www.dfdl.org/2003/primitives"
xmlns:dfdl="http://www.dfdl.org/2003/dfdl" xmlns:xs="http://www.w3.org/2001/XMLSchema"
xmlns:ns1="http://www.dfdl.org/2003/primitives" elementFormDefault="qualified"
attributeFormDefault="unqualified">
   <xs:import namespace="http://www.dfdl.org/2003/dfdl" schemaLocation="dfdl.xsd"/>
   <xs:attributeGroup name="primitiveAttributes">
      <xs:attribute name="byteOrder">
         <xs:simpleType>
            <xs:restriction base="xs:NMTOKEN">
               <xs:enumeration value="bigEndian"/>
               <xs:enumeration value="littleEndian"/>
```

```
            </xs:restriction>
         </xs:simpleType>
      </xs:attribute>
   </xs:attributeGroup>
   <xs:complexType name="typeOfPrimType" mixed="true">
      <xs:complexContent mixed="true">
         <xs:extension base="dfdl:typeOfType">
            <xs:attributeGroup ref="ns1:primitiveAttributes"/>
         </xs:extension>
      </xs:complexContent>
   </xs:complexType>
   <xs:element name="byte" type="ns1:typeOfPrimType" substitutionGroup="dfdl:type"/>
   <xs:element name="short" type="ns1:typeOfPrimType" substitutionGroup="dfdl:type"/>
   <xs:element name="int" type="ns1:typeOfPrimType" substitutionGroup="dfdl:type"/>
   <xs:element name="long" type="ns1:typeOfPrimType" substitutionGroup="dfdl:type"/>
   <xs:element name="char" type="ns1:typeOfPrimType" substitutionGroup="dfdl:type"/>
   <xs:element name="float" type="ns1:typeOfPrimType" substitutionGroup="dfdl:type"/>
   <xs:element name="double" type="ns1:typeOfPrimType" substitutionGroup="dfdl:type"/>
   <xs:element name="boolean" type="ns1:typeOfPrimType" substitutionGroup="dfdl:type"/>
   <xs:element name="digit" type="ns1:typeOfPrimType" substitutionGroup="dfdl:type"/>
   <xs:element name="letter" type="ns1:typeOfPrimType" substitutionGroup="dfdl:type"/>
   <xs:element name="alphanumeric" type="ns1:typeOfPrimType"
substitutionGroup="dfdl:type"/>
   <xs:element name="nonAlphanumeric" type="ns1:typeOfPrimType"
substitutionGroup="dfdl:type"/>
   <xs:element name="lineFeed" type="ns1:typeOfPrimType" substitutionGroup="dfdl:type"/>
   <xs:element name="carriageReturn" type="ns1:typeOfPrimType"
substitutionGroup="dfdl:type"/>
   <xs:element name="tab" type="ns1:typeOfPrimType" substitutionGroup="dfdl:type"/>
   <xs:element name="space" type="ns1:typeOfPrimType" substitutionGroup="dfdl:type"/>
   <xs:element name="whiteSpace" type="ns1:typeOfPrimType" substitutionGroup="dfdl:type"/>
   <xs:element name="fullStop" type="ns1:typeOfPrimType" substitutionGroup="dfdl:type"/>
   <xs:element name="comma" type="ns1:typeOfPrimType" substitutionGroup="dfdl:type"/>
   <xs:element name="null" type="ns1:typeOfPrimType" substitutionGroup="dfdl:type"/>
   <xs:element name="minusSign" type="ns1:typeOfPrimType" substitutionGroup="dfdl:type"/>
</xs:schema>
```

3. Structural definition

The structural definition is an XML document written in SDL using the SDL schema extended with the schema extensions above. It defines the structure of the new types:

In SDL formal language:

```
byte := ( bit.8 )
short := ( byte.2 )
int := ( byte.4 )
long := ( byte.8 )
char := byte
float := ( byte.4 )
double := ( byte.8 )
boolean := byte
digit := char[ '0' ] ~ char[ '9' ]
letter := ( char[ 'a' ] ~ char[ 'z' ] | char[ 'A' ] ~ char[ 'Z' ] )
alphanumeric := ( digit | letter )
nonAlphanumeric := char - alphanumeric
lineFeed := char[ '\n' ]
carriageReturn := char[ '\m' ]
space := char[ '\s' ]
tab := char[ '\t' ]
whiteSpace := ( lineFeed | carriageReturn | tab | space )
fullStop := char[ '.' ]
```

```
comma := char[ ',' ]
null := char[ '\0' ]
minusSign := char[ '-' ]
```

In XML:

```
<?xml version="1.0" encoding="UTF-8"?>
<?xml-stylesheet type="text/xsl" href="C:\Documents and Settings\martin\My
Documents\Grid\dfdl\Drafts\XML\xml2sdl.xsl"?>
<dfdl:dfdl xmlns="http://www.dfdl.org/2003/primitives"
xmlns:dfdl="http://www.dfdl.org/2003/dfdl" xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance" xsi:schemaLocation="http://www.dfdl.org/2003/primitives
primitives.xsd">
   <dfdl:definitions>
      <!--      -->
      <dfdl:define>
         <byte/>
         <dfdl:toBe>
            <dfdl:repeat number="8">
               <dfdl:bit/>
            </dfdl:repeat>
         </dfdl:toBe>
      </dfdl:define>
      <!--      -->
      <dfdl:define>
         <short/>
         <dfdl:toBe>
            <dfdl:repeat number="2">
               <byte/>
            </dfdl:repeat>
         </dfdl:toBe>
      </dfdl:define>
      <!--      -->
      <dfdl:define>
         <int/>
         <dfdl:toBe>
            <dfdl:repeat number="4">
               <byte/>
            </dfdl:repeat>
         </dfdl:toBe>
      </dfdl:define>
      <!--      -->
      <dfdl:define>
         <long/>
         <dfdl:toBe>
            <dfdl:repeat number="8">
               <byte/>
            </dfdl:repeat>
         </dfdl:toBe>
      </dfdl:define>
      <!--      -->
      <dfdl:define>
         <char/>
         <dfdl:toBe>
            <byte/>
         </dfdl:toBe>
      </dfdl:define>
      <!--      -->
      <dfdl:define>
         <float/>
         <dfdl:toBe>
            <dfdl:repeat number="4">
               <byte/>
```

```
      </dfdl:repeat>
   </dfdl:toBe>
</dfdl:define>
<!--      -->
<dfdl:define>
   <double/>
   <dfdl:toBe>
      <dfdl:repeat number="8">
         <byte/>
      </dfdl:repeat>
   </dfdl:toBe>
</dfdl:define>
<!--      -->
<dfdl:define>
   <boolean/>
   <dfdl:toBe>
      <byte/>
   </dfdl:toBe>
</dfdl:define>
<!--      -->
<dfdl:define>
   <digit/>
   <dfdl:toBe>
      <dfdl:range>
         <char>0</char>
         <char>9</char>
      </dfdl:range>
   </dfdl:toBe>
</dfdl:define>
<!--      -->
<dfdl:define>
   <letter/>
   <dfdl:toBe>
      <dfdl:either>
         <dfdl:range>
            <char>a</char>
            <char>z</char>
         </dfdl:range>
         <dfdl:range>
            <char>A</char>
            <char>Z</char>
         </dfdl:range>
      </dfdl:either>
   </dfdl:toBe>
</dfdl:define>
<!--      -->
<dfdl:define>
   <alphanumeric/>
   <dfdl:toBe>
      <dfdl:either>
         <digit/>
         <letter/>
      </dfdl:either>
   </dfdl:toBe>
</dfdl:define>
<!--      -->
<dfdl:define>
   <nonAlphanumeric/>
   <dfdl:toBe>
      <dfdl:exclude>
         <alphanumeric/>
         <dfdl:from>
            <char/>
```

```
         </dfdl:from>
      </dfdl:exclude>
   </dfdl:toBe>
</dfdl:define>
<!--      -->
<dfdl:define>
   <lineFeed/>
   <dfdl:toBe>
      <char>\n</char>
   </dfdl:toBe>
</dfdl:define>
<!--      -->
<dfdl:define>
   <carriageReturn/>
   <dfdl:toBe>
      <char>\m</char>
   </dfdl:toBe>
</dfdl:define>
<!--      -->
<dfdl:define>
   <space/>
   <dfdl:toBe>
      <char>\s</char>
   </dfdl:toBe>
</dfdl:define>
<!--      -->
<dfdl:define>
   <tab/>
   <dfdl:toBe>
      <char>\t</char>
   </dfdl:toBe>
</dfdl:define>
<!--      -->
<dfdl:define>
   <whiteSpace/>
   <dfdl:toBe>
      <dfdl:either>
         <lineFeed/>
         <carriageReturn/>
         <tab/>
         <space/>
      </dfdl:either>
   </dfdl:toBe>
</dfdl:define>
<!--      -->
<dfdl:define>
   <fullStop/>
   <dfdl:toBe>
      <char>.</char>
   </dfdl:toBe>
</dfdl:define>
<!--      -->
<dfdl:define>
   <comma/>
   <dfdl:toBe>
      <char>,</char>
   </dfdl:toBe>
</dfdl:define>
<!--      -->
<dfdl:define>
   <null/>
   <dfdl:toBe>
      <char>\0</char>
```

```
         </dfdl:toBe>
      </dfdl:define>
      <!--      -->
      <dfdl:define>
         <minusSign/>
         <dfdl:toBe>
            <char>-</char>
         </dfdl:toBe>
      </dfdl:define>
   </dfdl:definitions>
</dfdl:dfdl>
```

4. Basic API

The basic API section defines the behaviour of the returns for the basic API calls on each of the new objects.

Need guiding principles for this – I have assumed that the library will carry out trivial conversion/casting but we could insist this is done in the language first…thoughts?
Need to specify details of exceptions
No doubt descriptions will be longer this is for illustration
There will be mistakes in this I have not spent very long on these definitions
There is currently no notion of inheritance…clearly we want this to get sensible default behaviour…how to specify? (might make descriptions more contained?)

4.1 byte

call		semantic
byte	**getAsByte**();	return byte value converting for bit/byte order
short	**getAsShort**();	return byte value converting for bit/byte order
int	**getAsInt**();	return byte value converting for bit/byte order
long	**getAsLong**();	return byte value converting for bit/byte order
char	**getAsChar**();	return byte value converting for bit/byte order
float	**getAsFloat**();	return byte value converting for bit/byte order
double	**getAsDouble**();	return byte value converting for bit/byte order
boolean	**getAsBoolean**();	raise an exception
String	**getAsString**();	return string of the byte value as a decimal integer
void	**set** (byte value);	sets value of the byte
void	**set** (short value);	set value, raise an exception on overflow
void	**set** (int value);	set value, raise an exception on overflow
void	**set** (long value);	set value, raise an exception on overflow
void	**set** (char value);	set value of the byte to the numerical value of the char
void	**set** (float value);	raise an exception
void	**set** (double value);	raise an exception
void	**set** (boolean value);	raise an exception
void	**set** (String value);	convert as an integer, raise an exception on overflow
byte[]	**getAsByteArray**();	return an array of a single byte converting for bit/byte order
short[]	**getAsShortArray** ();	return an array of a single value converting for bit/byte order
int[]	**getAsInArrayt**();	return an array of a single value converting for bit/byte order
long[]	**getAsLongArray** ();	return an array of a single value converting for bit/byte order
char[]	**getAsCharArray** ();	return an array of a single value converting for bit/byte order
float[]	**getAsFloatArray** ();	return an array of a single value converting for bit/byte order
double[]	**getAsDoubleArray** ();	return an array of a single value converting for bit/byte order
boolean[]	**getAsBooleanArray** ();	raise an exception
String[]	**getAsStringArray** ();	return an array with a single string with the byte value as a decimal integer
void	**set** (byte[] value);	If array has a single element, set value else raise an exception
void	**set** (short[] value);	raise an exception
void	**set** (int[] value);	raise an exception
void	**set** (long[] value);	raise an exception
void	**set** (char[] value);	raise an exception
void	**set** (float[] value);	raise an exception
void	**set** (double[] value);	raise an exception
void	**set** (boolean[] value);	raise an exception
void	**set** (String[] value);	convert as an integer, raise an exception on overflow

4.2 short

call		semantic
byte	**getAsByte**();	raise exception
short	**getAsShort**();	return value converting for bit/byte order
int	**getAsInt**();	return value converting for bit/byte order
long	**getAsLong**();	return value converting for bit/byte order
char	**getAsChar**();	return value converting for bit/byte order
float	**getAsFloat**();	return value converting for bit/byte order
double	**getAsDouble**();	return value converting for bit/byte order
boolean	**getAsBoolean**();	raise an exception
String	**getAsString**();	return string of the byte value as a decimal integer
void	**set** (byte value);	set value
void	**set** (short value);	set value
void	**set** (int value);	set value, raise an exception on overflow
void	**set** (long value);	set value, raise an exception on overflow
void	**set** (char value);	set value, to the numerical value of the char
void	**set** (float value);	raise an exception
void	**set** (double value);	raise an exception
void	**set** (boolean value);	raise an exception
void	**set** (String value);	convert as an integer, raise an exception on overflow
byte[]	**getAsByteArray**();	returns array of underlying bytes
short[]	**getAsShortArray** ();	return an array of a single value converting for bit/byte order
int[]	**getAsInArrayt**();	return an array of a single value converting for bit/byte order
long[]	**getAsLongArray** ();	return an array of a single value converting for bit/byte order
char[]	**getAsCharArray** ();	return an array of a single value converting for bit/byte order
float[]	**getAsFloatArray** ();	return an array of a single value converting for bit/byte order
double[]	**getAsDoubleArray** ();	return an array of a single value converting for bit/byte order
boolean[]	**getAsBooleanArray** ();	raise an exception
String[]	**getAsStringArray** ();	return an array with a single string with the value as a decimal integer
void	**set** (byte[] value);	If the array has two elements set value, respecting current byteOrder setting else raise an exception
void	**set** (short[] value);	If the array has a single element set value else raise an exception
void	**set** (int[] value);	raise an exception
void	**set** (long[] value);	raise an exception
void	**set** (char[] value);	raise an exception
void	**set** (float[] value);	raise an exception
void	**set** (double[] value);	raise an exception
void	**set** (boolean[] value);	raise an exception
void	**set** (String[] value);	convert as an integer, raise an exception on overflow

4.3 int

call		semantic
byte	**getAsByte**();	raise an exception
short	**getAsShort**();	raise an exception
int	**getAsInt**();	return value converting for bit/byte order
long	**getAsLong**();	return value converting for bit/byte order
char	**getAsChar**();	return value converting for bit/byte order
float	**getAsFloat**();	return value converting for bit/byte order
double	**getAsDouble**();	return value converting for bit/byte order
boolean	**getAsBoolean**();	raise an exception
String	**getAsString**();	return string of the byte value as a decimal integer
void	**set** (byte value);	sets value
void	**set** (short value);	set value, raise an exception on overflow
void	**set** (int value);	set value, raise an exception on overflow
void	**set** (long value);	set value, raise an exception on overflow
void	**set** (char value);	set value to the numerical value of the char
void	**set** (float value);	raise an exception
void	**set** (double value);	raise an exception
void	**set** (boolean value);	raise an exception
void	**set** (String value);	convert as an integer, raise an exception on overflow
byte[]	**getAsByteArray**();	returns array of underlying bytes
short[]	**getAsShortArray** ();	raise an exception
int[]	**getAsIntArrayt**();	return an array of a single value converting for bit/byte order
long[]	**getAsLongArray** ();	return an array of a single value converting for bit/byte order
char[]	**getAsCharArray** ();	raise an exception
float[]	**getAsFloatArray** ();	return an array of a single value converting for bit/byte order
double[]	**getAsDoubleArray** ();	return an array of a single value converting for bit/byte order
boolean[]	**getAsBooleanArray** ();	raise an exception
String[]	**getAsStringArray** ();	return an array with a single string with value as a decimal integer
void	**set** (byte[] value);	If the array has four elements set value, respecting current byteOrder setting else raise an exception
void	**set** (short[] value);	raise an exception
void	**set** (int[] value);	If the array has a single element set value else raise an exception
void	**set** (long[] value);	raise an exception
void	**set** (char[] value);	raise an exception
void	**set** (float[] value);	raise an exception
void	**set** (double[] value);	raise an exception
void	**set** (boolean[] value);	raise an exception
void	**set** (String[] value);	convert as an integer, raise an exception on overflow

4.4 long

call		semantic
byte	**getAsByte**();	return value converting for bit/byte order
short	**getAsShort**();	return value converting for bit/byte order
int	**getAsInt**();	return value converting for bit/byte order
long	**getAsLong**();	return value converting for bit/byte order
char	**getAsChar**();	return value converting for bit/byte order
float	**getAsFloat**();	return value converting for bit/byte order
double	**getAsDouble**();	return value converting for bit/byte order
boolean	**getAsBoolean**();	raise an exception
String	**getAsString**();	return string value as a decimal integer
void	**set** (byte value);	sets value
void	**set** (short value);	set value
void	**set** (int value);	set value
void	**set** (long value);	set value
void	**set** (char value);	set value to the numerical value of the char
void	**set** (float value);	raise an exception
void	**set** (double value);	raise an exception
void	**set** (boolean value);	raise an exception
void	**set** (String value);	convert as an integer, raise an exception on overflow
byte[]	**getAsByteArray**();	returns array of underlying bytes
short[]	**getAsShortArray** ();	raise an exception
int[]	**getAsInArrayt**();	raise an exception
long[]	**getAsLongArray** ();	return an array of a single value converting for bit/byte order
char[]	**getAsCharArray** ();	return an array of a single value converting for bit/byte order
float[]	**getAsFloatArray** ();	return an array of a single value converting for bit/byte order
double[]	**getAsDoubleArray** ();	return an array of a single value converting for bit/byte order
boolean[]	**getAsBooleanArray** ();	raise an exception
String[]	**getAsStringArray** ();	return an array with a single string with the byte value as a decimal integer
void	**set** (byte[] value);	If the array has 8 elements set value, respecting current byteOrder setting else raise an exception
void	**set** (short[] value);	raise an exception
void	**set** (int[] value);	raise an exception
void	**set** (long[] value);	If the array has a single element set value else raise an exception
void	**set** (char[] value);	raise an exception
void	**set** (float[] value);	raise an exception
void	**set** (double[] value);	raise an exception
void	**set** (boolean[] value);	raise an exception
void	**set** (String[] value);	convert as an integer, raise an exception on overflow

4.5 char

call		semantic
byte	**getAsByte**();	return value converting for bit/byte order
short	**getAsShort**();	return value converting for bit/byte order
int	**getAsInt**();	return value converting for bit/byte order
long	**getAsLong**();	return value converting for bit/byte order
char	**getAsChar**();	return value converting for bit/byte order
float	**getAsFloat**();	return value converting for bit/byte order
double	**getAsDouble**();	return value converting for bit/byte order
boolean	**getAsBoolean**();	raise an exception
String	**getAsString**();	convert to a string containing the character
void	**set** (byte value);	sets value
void	**set** (short value);	set value, raise an exception on overflow
void	**set** (int value);	set value, raise an exception on overflow
void	**set** (long value);	set value, raise an exception on overflow
void	**set** (char value);	set value
void	**set** (float value);	raise an exception
void	**set** (double value);	raise an exception
void	**set** (boolean value);	raise an exception
void	**set** (String value);	If the string contains a single character, use it otherwise raise an exception
byte[]	**getAsByteArray**();	return an array of a single value converting for bit/byte order
short[]	**getAsShortArray** ();	return an array of a single value converting for bit/byte order
int[]	**getAsInArrayt**();	return an array of a single value converting for bit/byte order
long[]	**getAsLongArray** ();	return an array of a single value converting for bit/byte order
char[]	**getAsCharArray** ();	return an array of a single value converting for bit/byte order
float[]	**getAsFloatArray** ();	return an array of a single value converting for bit/byte order
double[]	**getAsDoubleArray** ();	return an array of a single value converting for bit/byte order
boolean[]	**getAsBooleanArray** ();	raise an exception
String[]	**getAsStringArray** ();	return an array with a single string with the byte value as a decimal integer
void	**set** (byte[] value);	If the array has a single element set value else raise an exception
void	**set** (short[] value);	raise an exception
void	**set** (int[] value);	raise an exception
void	**set** (long[] value);	raise an exception
void	**set** (char[] value);	If the array has a single element set value else raise an exception
void	**set** (float[] value);	raise an exception
void	**set** (double[] value);	raise an exception
void	**set** (boolean[] value);	raise an exception
void	**set** (String[] value);	If there is one string and it contains one character use it other, raise an exception

4.6 float

call		semantic
byte	**getAsByte**();	raise exception
short	**getAsShort**();	raise exception
int	**getAsInt**();	raise exception
long	**getAsLong**();	raise exception
char	**getAsChar**();	raise exception
float	**getAsFloat**();	return value as IEEE floating point respecting bit/byte order
double	**getAsDouble**();	return value as IEEE floating point respecting bit/byte order
boolean	**getAsBoolean**();	raise exception
String	**getAsString**();	return string value of a floating point
void	**set** (byte value);	sets value as an integer
void	**set** (short value);	set value
void	**set** (int value);	set value
void	**set** (long value);	set value
void	**set** (char value);	set value
void	**set** (float value);	set value
void	**set** (double value);	set value, raise exception on overflow
void	**set** (boolean value);	raise an exception
void	**set** (String value);	convert as a float, raise an exception on overflow
byte[]	**getAsByteArray**();	return an array of the underlying bytes
short[]	**getAsShortArray** ();	raise exception
int[]	**getAsInArrayt**();	raise exception
long[]	**getAsLongArray** ();	raise exception
char[]	**getAsCharArray** ();	raise exception
float[]	**getAsFloatArray** ();	return an array with a single value
double[]	**getAsDoubleArray** ();	return an array with a single value
boolean[]	**getAsBooleanArray** ();	raise an exception
String[]	**getAsStringArray** ();	return an array with a single string containing a representation of the value
void	**set** (byte[] value);	set the underlying bytes
void	**set** (short[] value);	raise an exception
void	**set** (int[] value);	raise an exception
void	**set** (long[] value);	raise an exception
void	**set** (char[] value);	raise an exception
void	**set** (float[] value);	raise an exception
void	**set** (double[] value);	raise an exception
void	**set** (boolean[] value);	raise an exception
void	**set** (String[] value);	convert as an float, raise an exception on overflow

4.7 double

call		semantic
byte	**getAsByte**();	raise exception
short	**getAsShort**();	raise exception
int	**getAsInt**();	raise exception
long	**getAsLong**();	raise exception
char	**getAsChar**();	raise exception
float	**getAsFloat**();	raise exception
double	**getAsDouble**();	return value as IEEE floating point respecting bit/byte order
boolean	**getAsBoolean**();	raise exception
String	**getAsString**();	return string value of a floating point
void	**set** (byte value);	sets value as an integer
void	**set** (short value);	set value
void	**set** (int value);	set value
void	**set** (long value);	set value
void	**set** (char value);	set value
void	**set** (float value);	set value
void	**set** (double value);	set value
void	**set** (boolean value);	raise an exception
void	**set** (String value);	convert as a float, raise an exception on overflow
byte[]	**getAsByteArray**();	return an array of the underlying bytes
short[]	**getAsShortArray** ();	raise exception
int[]	**getAsInArrayt**();	raise exception
long[]	**getAsLongArray** ();	raise exception
char[]	**getAsCharArray** ();	raise exception
float[]	**getAsFloatArray** ();	return an array with a single value
double[]	**getAsDoubleArray** ();	return an array with a single value
boolean[]	**getAsBooleanArray** ();	raise an exception
String[]	**getAsStringArray** ();	return an array with a single string containing a representation of the value
void	**set** (byte[] value);	set the underlying bytes, exception if there is the wrong number
void	**set** (short[] value);	raise an exception
void	**set** (int[] value);	raise an exception
void	**set** (long[] value);	raise an exception
void	**set** (char[] value);	raise an exception
void	**set** (float[] value);	raise an exception
void	**set** (double[] value);	raise an exception
void	**set** (boolean[] value);	raise an exception
void	**set** (String value);	convert as an float, raise an exception on overflow

4.8 boolean

call		semantic
byte	**getAsByte**();	return underlying byte representation
short	**getAsShort**();	raise an exception
int	**getAsInt**();	raise an exception
long	**getAsLong**();	raise an exception
char	**getAsChar**();	raise an exception
float	**getAsFloat**();	raise an exception
double	**getAsDouble**();	raise an exception
boolean	**getAsBoolean**();	return value
String	**getAsString**();	return string containing “true” or “false”
void	**set** (byte value);	raise an exception
void	**set** (short value);	raise an exception
void	**set** (int value);	raise an exception
void	**set** (long value);	raise an exception
void	**set** (char value);	raise an exception
void	**set** (float value);	raise an exception
void	**set** (double value);	raise an exception
void	**set** (boolean value);	set value
void	**set** (String value);	convert from containing “true/True/TRUE” or “false/False/FALSE” raise exception otherwise
byte[]	**getAsByteArray**();	return an array of a single byte
short[]	**getAsShortArray** ();	raise an exception
int[]	**getAsInArrayt**();	raise an exception
long[]	**getAsLongArray** ();	raise an exception
char[]	**getAsCharArray** ();	raise an exception
float[]	**getAsFloatArray** ();	raise an exception
double[]	**getAsDoubleArray** ();	raise an exception
boolean[]	**getAsBooleanArray** ();	return an array with a single value
String[]	**getAsStringArray** ();	return an array with a single string with the value as “true” or “false”
void	**set** (byte[] value);	set underlying bytes
void	**set** (short[] value);	raise an exception
void	**set** (int[] value);	raise an exception
void	**set** (long[] value);	raise an exception
void	**set** (char[] value);	raise an exception
void	**set** (float[] value);	raise an exception
void	**set** (double[] value);	raise an exception
void	**set** (boolean[] value);	If array has a single element set to this value otherwise raise an exception
void	**set** (String[] value);	If array has a single element set to this value otherwise raise an exception

4.9 digit, letter, alphanumeric, nonAlphanumeric, linefeed, carriageReturn, tab, whiteSpace, comma, fullStop, null, minusSign

These types are all strict subsets of char. They have essentially the same behaviour as char except that they will raise an exception if an attempt is made to set their value to any other than the values they are permitted to have.

Author Information

Martin Westhead, M.Westhead@epcc.ed.ac.uk, EPCC, University of Edinburgh. James Clerk Maxwell Building, Mayfield Road, Edinburgh EH9 3JZ, UK.

Glossary

DFDL – Data Format Description Language

Intellectual Property Statement

Full Copyright Notice

References

BinX http://www.epcc.ed.ac.uk/gridserve/WP5/Binx/
HDF http://hdf.ncsa.uiuc.edu/HDF5
BDF/SAM http://collaboratory.emsl.pnl.gov/docs/collab/sam
XDR http://www.faqs.org/rfcs/rfc1014.html
DFDL web pages http://www.epcc.ed.ac.uk/dfdl