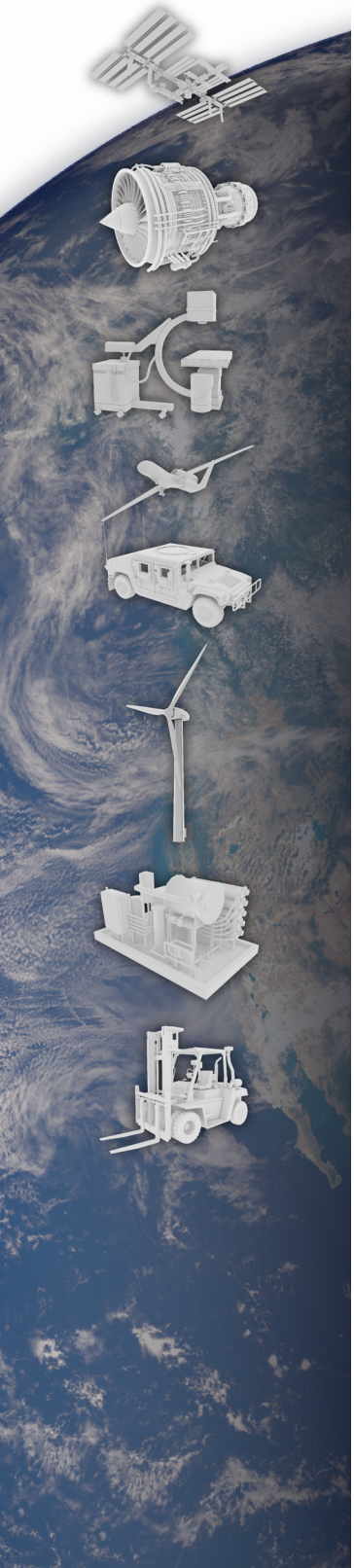

CoreDX DDS

Type System

Designing and Using Data Types

with

X-Types and IDL 4

TWINOAKS COMPUTING INC.

PRACTICAL MIDDLEWARE EXPERTISE

2017-01-23

Table of Contents

1 Index of Tables

1 Introduction

The **Extensible and Dynamic Types for DDS** (X-Types) standard provides a robust and flexible specification of the data types for use in the Data Distribution Service (DDS) middleware. In conjunction with the IDL 4 specification, the X-Types standard introduces several new concepts for data types that enhance the options for defining, discovering, extending, and interacting with application defined data types.

There are many motivations for the X-Types and IDL 4 update. First, the update formally defines the data type system used by DDS; this includes the full range of available data types, interfaces, and data augmentation (for example, keys). Second, the type system supports type evolution and type inheritance; important concepts for large and long-lived systems. Third, the standard includes a full API for the DynamicType and DynamicData concepts; providing the ability to define types at run-time. Finally, the system adds a few new data types that were missing from traditional IDL.

This document presents the full data type system available to CoreDX DDS users, and then explores some advanced data type designs that demonstrate the use of the new features available with IDL 4 and X-Types.

1.1 Overview

The X-Types standard includes the primitive and constructed types from the previous IDL based type system. In addition, the standard introduces two new types: **map** and **bitmask**. The '**map**' type represents a collection of pairs. A pair is composed of a 'key' and a 'value'. The map provides the ability to retrieve a 'value' given a 'key'. The **bitmask** type represents an ordered set of boolean flags. This type allows for more compact representation of a set of boolean flags. The **string** types are composed of characters of either 8 or 32 bits (string and wstring, respectively).

Some types that existed previously have been enhanced. For example, the **enum** type can now specify its size: from 1 to 32 bits. Previously, enum instances were all 32 bits.

The IDL 4.0 and X-Types standards build upon the previously used Interface Definition Language (IDL) as a language for type definition. It defines some small modifications to the existing IDL grammar to support the new types. Further, the X-Types standard introduces an XML grammar that can be used to define data types. A developer can choose to use either IDL or XML based type definition.

To support RPC over DDS, the syntax for **interface** definition is included in the subset of IDL 4.0 that is supported by the CoreDX DDS IDL parser.

The type system introduces support for associating additional or auxiliary information with a type. The mechanism for this is the 'annotation' construct.

The type system includes support for single inheritance (structures can inherit from other structures).

The standard includes rules for determining compatibility between two types. These rules are useful when determining compatibility between a DataWriter and DataReader.

The standard defines a new mechanism for encoding data on the wire. This data encoding is an enhancement of the Common Data Representation (CDR) used in prior DDS implementations. It supports the concept of 'optional' data members, and modified member ordering. These features allow the type designer to craft sophisticated data types that may be more compact on the network or support type evolution in a large system.

Finally, the X-Types standard defines an API for programmatically defining data types (DynamicType) and creating instances (DynamicData).

2 Type Definition

Types can be defined in one of three ways: IDL, XML, and programmatically with DynamicType. A type defined using one mechanism is identical to the same type defined in another mechanism. [That is, the definition mechanism has no bearing on the type itself.]

The CoreDX DDS code generator (coredx_ddl) accepts IDL or XML as input for type definition. This section describes the supported syntax. In comparison to previous versions of CoreDX DDS, the traditional IDL syntax has been augmented to support new X-Types concepts (annotations) and types

(bitmask and map). Also, the support for XML type definition is new for CoreDX DDS version 4.0.

2.1 Primitive Types

The following primitive types are supported by the type definition language:

IDL type	XML Element	C	C++	C#	Java
char	char8	char	char	char	char
wchar	char32	char32	char32	int	int
octet	byte	unsigned char	unsigned char	byte	byte
boolean	boolean	unsigned char	unsigned char	bool	boolean
short	int16	short	short	short	short
unsigned short	uint16	unsigned short	unsigned short	ushort	short
long	int32	int	int	int	int
unsigned long	uint32	unsigned int	unsigned int	uint	int
long long	int64	int64_t	int64_t	long	long
unsigned long long	uint64	uint64_t	uint64_t	ulong	long
float	float32	float	float	float	float
double	float64	double	double	double	double

Table 1: Primitive Types

If the primitive type is annotated as 'external' or 'optional', then the language mapping is modified to be a 'pointer', 'box', or nullable type as shown in the following table.

'external | optional' IDL type	'external | optional' XML Element	C	C++	C-Sharp	Java
char	char8	char *	char *	char?	Character
wchar	char32	char32 *	char32 *	int?	Integer
octet	byte	unsigned char *	unsigned char *	byte?	Byte
boolean	boolean	unsigned char *	unsigned char *	bool?	Boolean
short	int16	short *	short *	short?	Short
unsigned short	uint16	unsigned short *	unsigned short *	ushort?	Short
long	int32	int *	int *	int?	Integer
unsigned long	uint32	unsigned int *	unsigned int *	uint?	Integer
long long	int64	int64_t *	int64_t *	long?	Long
unsigned long long	uint64	uint64_t *	uint64_t *	ulong?	Long
float	float32	float *	float *	float?	Float
double	float64	double *	double *	double?	Double

Table 2: Primitive Types [optional or external]

2.2 Collection Types

The following sections identify the syntax and language mappings of the “collection” types. These types include enumerations, bitmasks, strings, arrays, sequences and maps.

2.2.1 Enumeration Types

An enumeration type includes a 'name' and a list of named constants with optional assigned values. In IDL this is specified like this:

```
enum EnumName { CONST1, CONST2, CONST3 };
```

This type is expressed in XML this way:

```
<enum name=”EnumName” bitBound=”32”>
      <enumerator name=”CONST1” />
      <enumerator name=”CONST2” />
      <enumerator name=”CONST3” />
</enum>
```

This type defines an enumeration named 'EnumName' with three named constants 'CONST1' (etc). The example definition does not assign values to the constants, so they are assigned automatically, using a 1-up counter starting at zero. So, CONST1 = 0, CONST2 = 1, etc.

```
enum EnumName2 { C1 = 1, C2 = 2, C3 = 3 };
```

In this definition, the constant values are specified explicitly. An alternative and equivalent syntax, using annotations is:

```
enum EnumName2 {
      @value(1) C1,              /* using 'pre annotation'  */
                C2, //@value(2)  /* using 'post annotation' */
      @value(3) C3
};
```

And the same type in XML syntax:

```
<enum name="EnumName2" bitBound="32">
  <enumerator name="C1" value=”1”/>
  <enumerator name="C2" value="2"/>
  <enumerator name="C3" value=”3”/>
</enum>
```

Further, if some values are specified and other values are left to the 'default' algorithm, then the default algorithm will assign a value based on the 'largest seen so far' value plus one. So,

```
enum EnumName3 {
      @value(10) A1, /* is '10' */
                 A2, /* is '11' */
      @value(9)  A3, /* is '9'  */
                 A4  /* is '12' */
};
```

It is an error for an enumeration to contain two or more named constants with the same value or the same name.

The 'size' of an instance of an enumeration type in all language mappings is 32 bits by default. This can be adjusted by applying the 'BitBound' annotation. For example, in IDL:

```
@bit_bound(8)
enum SmallEnum { A, B, C };
```

And in XML:

```
<enum name="SmallEnum" bitBound="8">
  <enumerator name="A" />
  <enumerator name="B" />
  <enumerator name="C" />
</enum>
```

This will cause the enumeration type to be mapped to a smaller data type (for example, unsigned short in the 'C' language mapping).

2.2.1.1 C Language Mapping

In C, an enum is mapped as a typedef. The enumerated constants are mapped to #define statements. For example:

```
enum A { A1, A2, A3 };
```

Maps to

```
typedef unsigned int A;
#define A1      0
#define A2      1
#define A3      2
```

The size of the type in the typedef is adjusted by applying a 'BitBound' annotation. The type is 'unsigned short' if 1<bound<=16, and uint32_t if 16<bound<=32. Without the BitBound annotation, the bound is set to the default of 32 bits.

2.2.1.2 C++ Language Mapping

The C++ mapping is to an 'enum' type declaration. For example:

```
enum A { A1, A2, A3 };
```

maps to

```
enum A { //unsigned int
  A1 = 0,
  A2 = 1,
  A3 = 2 };
```

NOTE: unlike the C mapping, the C++ mapping is always to an 'enum' type and does not change based on any BitBound annotation. Note however, the encoding for transmission on the wire will honor any BitBound direction, making the data encoding compatible independent of the language mapping.

2.2.1.3 C# Language Mapping

Enumerations in C# map to an 'enum'. For example,

```
enum A { A1, A2, A3 };
```

maps to the following:

```
public enum A : uint {
  A1 = 0,
  A2 = 1,
  A3 = 2,
}
```

2.2.1.4 Java Language Mapping

In Java, an enumeration type is mapped to a class. The class includes public static final constants that represent the defined integral values of the enumeration; and, public static final instances of the class, one for each defined enumeration value. It also includes methods to convert from an 'integral' value to an instance of the enumeration class.

For example,

enum A { A1, A2, A3 };

maps to:

```
public class A {

  public static final int _A1 = 0;
  public static final A A1 = new A(_A1);
  public static final int _A2 = 1;
  public static final A A2 = new A(_A2);
  public static final int _A3 = 2;
  public static final A A3 = new A(_A3);
  public static final A __BAD_VALUE = new A(2);


  public int value() {
    return __pval;
  }

  public static A from_int(int v) {
    switch(v) {
      case 0:  return A1;
      case 1:  return A2;
      case 2:  return A3;
    }
    return __BAD_VALUE;
  }

  protected A(int v) {
    __pval = v;
  }

  private int __pval;
};
```

The enumeration values can be used like this:

```
   if (var == A.A1) ….
```

2.2.2 BitMask Types

Similar to the enumeration type, the **bitmask** consists of a 'name' and a list of named constants. The constants represent distinct flags with in the BitMask. The constants may be assigned values either manually or automatically.

There are two ways to specify a BitMask type in CoreDX DDS IDL. The first uses an annotation to coerce an Enumeration type into a BitMask type. The second approach treats **bitmask** as a 'first-class' type. The following two examples show each approach, and result in an equivalent type definition for 'MyBitmask'.

An IDL example bitmask (two equivalent types demonstrating the two alternative syntax forms):

```
@bitmask
enum MyBitmask { FLAG0, FLAG1, FLAG2 };

bitmask MyBitmask { FLAG0, FLAG1, FLAG2 };
```

The same type in XML:

```
<bitmask name="MyBitmask" bitBound="32">
  <flag name="FLAG0" />
  <flag name="FLAG1" />
  <flag name="FLAG2" />
</bitmask>
```

The values of the BitMask named constants represent the 'bit position' of the flag. For example, FLAG0 refers to bit '0' and FLAG1 refers to bit '1'.

The size of a BitMask may be specified with the BitBound annotation; it defaults to 32. It is an error for any named constant to have a value less than zero or beyond the range of the BitMask's BitBound. It is an error for any named constant to have a value equal to any other named constant within the BitMask.

The language mapping specifies that the named constants are mapped to the value 2^n where n = bit position. So, in the above example, FLAG0 is mapped to the value 1 (2^0), FLAG1 is mapped to 2 (2^1), and FLAG2 is mapped to 4 (2^2).

2.2.2.1 C Language Mapping

In C, a **bitmask** is mapped as a typedef. The boolean flag constants are mapped to #define statements. For example:

```
bitmask A { A1, A2, A3 };
```

maps to the following:

```
typedef unsigned int A;
typedef unsigned int ABits;
#define A1      1
#define A2      2
#define A3      4
```

The size of the type in the typedef is adjusted by applying a 'BitBound' annotation. The type is 'unsigned char' if 1<bound<=8; 'unsigned short' if 8<bound<=16, and uint32_t if 16<bound<=32; and uint64_t if 32<bound<=64. The maximum value for bound is 64. Without the BitBound annotation, the bound is set to the default of 32 bits.

2.2.2.2 C++ Language Mapping

The C++ bitmask mapping is similar to that of C, with the exception of the '#define' constants. In C++ these are replace with an 'enum' with the named constants as members. For example:

```
      bitmask A { A1, A2, A3 };
```

maps to the following:

```
typedef unsigned int A;
enum ABits { //unsigned int
  A1 = 1,
  A2 = 2,
```

```
  A3 = 4 };
```

2.2.2.3 C# Language Mapping

In C#, a bitmask type maps to an integral type that includes enough bits to accommodate the 'bit_bound' of the bitmask. An enum is generated that includes the defined bit flags. For example,

```
bitmask A { A1, A2, A3 };
```

is mapped to the following:

```
public enum A : uint {
  A1 = 1,
  A2 = 2,
  A3 = 4
}
```

And, a member of type bitmask 'A' is mapped to:

```
public uint a_bitmask_member;
```

2.2.2.4 Java Language Mapping

In Java, a **bitmask** type maps to an integral type that includes enough bits to accommodate the 'bit_bound' of the bitmask. Additionally, a class is generated that includes the defined bit flags. The class includes integral constants and class instances for each of the defined bit flags. The generated class name is the type name with “Bits” appended. For example,

```
bitmask A { A1, A2, A3 };
```

is mapped to the following:

```
public class ABits {

  public static final int _A1 = 1;
  public static final ABits A1 = new ABits(_A1);
  public static final int _A2 = 2;
  public static final ABits A2 = new ABits(_A2);
  public static final int _A3 = 4;
  public static final ABits A3 = new ABits(_A3);


  protected ABits(int v) {
    __pval = v;
  }

  private int __pval;
};
```

2.2.3 Array Types

Arrays are defined by applying a subscript notation to a symbol. It is common, but not required, to

specify an alias for an array type. Without an alias, the array type is declared 'anonymously' inside of another type (a structure, for example).

Here is an example of an array alias in IDL:

```
typedef long ArrayOfLong[10];
```

And, the same typedef in XML:

```
<typedef name="ArrayOfLong"
         type="int32"
         arrayDimensions="10" />
```

And, here is an IDL example of an 'anonymous' array type:

```
struct A {
  long array_of_long[10];
};
```

The same type in XML:

```
<struct name="A">
  <member name="array_of_long"
          id="0"
          type="int32"
          arrayDimensions="10"/>
</struct>
```

Arrays may have multiple dimensions. In XML:

```
typedef long Matrix2D[4][4];
```

And in XML:

```
<typedef name="Matrix2D"
         type="int32"
         arrayDimensions="4, 4" />
```

2.2.3.1 C Language Mapping

The above IDL (struct A) with a member of type long[10] maps to C like this:

```
typedef struct A {
   int  array_of_long [10];
} A;
```

2.2.3.2 C++ Language Mapping

In C++, the IDL maps to the following:

```
struct A {
   int  array_of_long [10];
};
```

2.2.3.3 C# Language Mapping

And, in C#, the IDL maps to the following:

```
public class A {
   public int[]  array_of_long;
}
```

The

2.2.3.4 Java Language Mapping

And, in Java:

```
public class A {
   public int[]  array_of_long;
};
```

2.2.4 String Types

Strings are considered collections of characters. The 'string' type is composed of 8-bit characters. The 'wstring' type is composed of 32-bit characters. A string may be bounded or unbounded. Unbounded strings have no upper limit placed on their data by the type system. [The runtime environment will necessarily impose a limit related to available memory.] The 'bound' does not included any required string 'termination' required by the language mapping. For example, the C language binding of a string with bound=5 will support 5 8-bit characters plus the **nul** termination byte for a total capacity of 6 bytes.

The keyword 'string' or 'wstring' introduces a String type, followed optionally by a length bound in angle brackets. The following IDL examples present bounded and unbounded string definitions:

```
string     an_unbounded_string;
string<10> a_fixed_10_string;
```

And in XML:

```
<typedef name="an_unbounded_string"
         type="string" />
<typedef name="a_fixed_10_string"
         type="string"
         stringMaxLength="10" />
```

2.2.4.1 C Language Mapping

Unbounded string is mapped to a “char *”. An unbounded wstring is mapped to a “cdx_char32_t *”. [cdx_char32_t is an alias for int32_t.] In either case, the user is responsible for providing a valid pointer to a region of characters that is **nul** terminated. It is expected that the user will allocate memory with CoreDX_DDS_malloc(). This is important if the string is part of a structure or union, because the clear() method of the object may attempt to reclaim the memory of its members using the corresponding CoreDX_DD_free() call. Alternatively, the user can reclaim the memory manually and set the pointer to NULL before invoking the type's clear() method.

A bounded string is mapped to an array of char with size equal to bound + 1. A bounded wstring is mapped to an array of cdx_char32_t elements with size equal to bound + 1. In this case, the user is responsible for initializing the array contents with valid character data including a **nul** termination character.

2.2.4.2 C++ Language Mapping

Unbounded string is mapped to a “char *”. An unbounded wstring is mapped to a “cdx_char32_t *”. [cdx_char32_t is an alias for int32_t.] In either case, the user is responsible for providing a valid pointer to a region of characters that is **nul** terminated. It is expected that the user will allocate memory with new[]. This is important if the string is part of a structure or union, because the clear() or destructor method of the object may attempt to reclaim the memory of its members. Alternatively, the user can reclaim the memory manually and set the pointer to NULL before invoking any clear or destroy methods.

A bounded string is mapped to an array of char with size equal to bound + 1. A bounded wstring is mapped to an array of cdx_char32_t elements with size equal to bound + 1. In this case, the user is responsible for initializing the array contents with valid character data including a **nul** termination character.

2.2.4.3 C# Language Mapping

In C#, string types map to String. This is true regardless of whether the string is bounded or unbounded.

For example, the string members like this:

```
struct Strings {
      string     an_unbounded_string;
      string<10> a_fixed_10_string;
      …
```

map to the following:

```
  public String an_unbounded_string;
  public String a_fixed_10_string;
```

2.2.4.4 Java Language Mapping

In Java, string types map to String. This is true regardless of whether the string is bounded or unbounded.

For example, the string members like this:

```
struct Strings {
      string     an_unbounded_string;
      string<10> a_fixed_10_string;
      …
```

map to the following:

```
public String an_unbounded_string;
public String a_fixed_10_string;
```

2.2.5 Sequence Types

Sequences provide an ordered collection of zero or more elements, where each element is of the same type. The sequence may have a defined upper bound on the number of elements, or may be 'unbounded'. Similar to arrays, sequence types may be named with a 'typedef' alias, or may be anonymous by declaring in-line within a containing type.

The IDL syntax for sequences is introduced with the 'sequence' keyword. Then, the type and an optional length bound is specified in angle brackets (separated with a comma, if the bound is provided). The bound is any 'constant' integral type, including an integral expression. For example, two sequence members can be declared like this:

```
sequence<string>   str_seq;
sequence<long,10>  long_10_seq;
```

The equivalent XML syntax for two sequence members is as follows:

```
<member name=”str_seq”
          id=”0”
          type=”string”
          sequenceMaxLength=”(-1)” />
<member name=”long_10_seq”
          id=”1”
          type=”long”
          sequenceMaxLength=”10” />
```

2.2.5.1 C Language Mapping

The C mapping of sequences use a macro to declare a small structure that contains members that implement the sequence. There are a collection of C functions that operate on these sequence structures, providing operations like 'add'. See the include file '**dds/dds_seq.h**' for the complete set of sequence operations.

In the C mapping, there is no difference between a bounded and unbounded sequence. Note, however, that the bound does impact the encode and decode behavior. If a user inserts more elements into a bounded sequence than allowed, the extra elements will not be transferred over the network, and they will not be available to a receiving DataReader.

For example:

```
typedef sequence<string>     Sequence_Unbounded_OfStrings;
typedef sequence<long, 10>   Sequence_10_OfLongs;
```

Maps to:

```
DECLARE_SEQ( char *, Sequence_Unbounded_OfString );
DECLARE_SEQ( int, Sequence_10_OfLong );
```

The mapping for an 'anonymous' sequence involves generating a name for the type. The generated type name is constructed by concatenating the containing scope(s), the fully-scoped sequence element type, and the suffix 'Seq'. It is possible that the generated name will clash with other auto-generated sequence type names. [For this reason, it is recommended that the user consider avoiding 'anonymous' sequences.]

2.2.5.2 C++ Language Mapping

The C++ sequence mapping utilizes a template type, and a macro to instantiate the template.

```
DECLARE_CPP_UNBOUNDED_SEQ( char *, Sequence_Unbounded_OfString );
DECLARE_CPP_UNBOUNDED_SEQ( int, Sequence_10_OfLong );
```

See the header file '**dds/dds_seq.hh**' for full details of the C++ sequence API.

2.2.5.3 C# Language Mapping

The C# mapping of bounded and unbounded sequences is to a simple array type. This IDL

```
struct A {
   sequence<string> seq_of_strings;
   sequence<long, 10> seq10_of_longs;
};
```

maps to the following members:

```
  public String[] seq_of_strings;
  public int[] seq_of_longs;
```

If the sequence is bounded, and the application provides an array with more elements than the upper bound, then the middleware will transmit only those elements within the bound.

2.2.5.4 Java Language Mapping

The Java mapping of bounded and unbounded sequences is to a simple array type. This IDL

```
struct A {
   sequence<string> seq_of_strings;
   sequence<long, 10> seq10_of_longs;
};
```

maps to the following members:

```
  public String[] seq_of_strings;
  public int[] seq_of_longs;
```

If the sequence is bounded, and the application provides an array with more elements than the upper bound, then the middleware will transmit only those elements within the bound.

2.2.6 Map Types

The **Map** type provides an associative mapping between a key and a value. When declaring a Map

type, the user specifies the type of the 'key' and the type of the value, and an optional upper bound on the number of key:value pairs. The user can insert key:value pair[s], and can lookup a 'key' to obtain the matching 'value'. The 'key' must be unique within the map. The types that can be used a Map key type are limited to signed and unsigned integer types, as well as string and wstring types.

Like other collection types (array, sequence, etc), the map type can be named using a 'typedef', or can be used in-place in which case, a type name is automatically generated.

Example IDL map definitions:

```
typedef  map<long,string>    Map_Unbounded_Long_String;
typedef  map<long,string,10> Map_10_Long_String;
```

Equivalent XML map definitions:

```
<typedef name="Map_Unbounded_Long_String"
         mapKeyType="int32"
         type="string"
         mapMaxLength="(-1)" />
<typedef name="Map_10_Long_String"
         mapKeyType="int32"
         type="string"
         mapMaxLength="10" />
```

2.2.6.1 C Language Mapping

The following 'map' IDL statements:

```
typedef<long,string>    Map_Unbounded_Long_String;
typedef map map<long,string,10> Map_10_Long_String;
```

Map to:

```
DDS_MAP_DECLARE(int, char *, map_int_string);
typedef map_int_string Map_Unbounded_Long_String;

DDS_MAP_DECLARE(int, char *, map_10_int_string);
typedef map_10_int_string Map_10_Long_String;
```

The 'DDS_MAP_DECLARE' macro defines a structure that holds the map members. These structures are operated on by a collection of functions declared in the **dds/dds_map.h** header file. The operations include clear(), get_size(), get_capacity(), set_size(), set_capacity(), copy(), insert(), find(), and replace().

2.2.6.2 C++ Language Mapping

The C++ mapping is very similar to the C mapping except that the map implementation is a template class. The above IDL map statements map to C++ like this:

```
DDS_CPP_MAP_DECLARE(int, char *, coredx_map_compare_int32, map_int_string);
typedef map_int_string Map_Unbounded_Long_String; /* ns:  */

DDS_CPP_MAP_DECLARE(int, char *, coredx_map_compare_int32, map_10_int_string);
typedef map_10_int_string Map_10_Long_String; /* ns:  */
```

2.2.6.3 C# Language Mapping

The C# mapping of the **map** type is to a generic collection Dictionary. For example:

```
public Dictionary<int, String> map_long_string;
public Dictionary<int, String> map_10_long_string;
```

As shown above, bounded maps are mapped identically unbounded maps. If the map contains more elements than indicated by the bound, then these elements are not transmitted by the write operation.

2.2.6.4 Java Language Mapping

The Java mapping is to the java.util.Map type. Internally, the generated code instantiates a java.util.HashMap to hold the map data.

For example:

```
public Map<Integer, String> map_long_string;
public Map<Integer, String> map_10_long_string;
```

As shown above, bounded maps are mapped identically unbounded maps. If the map contains more elements than indicated by the bound, then these elements are not transmitted by the write operation.

2.3 Aggregate Types

2.3.1 Structure Types

Structure types are an aggregate type that contains one or more member elements. The members can be of any type, including 'struct'. Structure members have a specific order, as listed in the IDL.

Structures can 'inherit' from another structure type. In this case, the child structure is said to 'extend' the parent structure. The parent structure must have been defined prior to the child.

2.3.1.1 Structure Properties

2.3.1.1.1 Extensibility

A structure has an 'extensibility'. There are three different kinds of extensibility: FINAL, EXTENSIBLE, and MUTABLE. The type designer can specify the extensibility of a structure through the use of the @extensibility annotation. If not present, then the extensibility defaults to 'EXTENSIBLE'. It is possible to change this default with a command-line option to the coredx_ddl code generator. [See the section on CoreDX DDL Compiler.]

The 'extensibility' primarily impacts the logic of 'matching' types between a DataReader and a DataWriter. With FINAL extensibility, the types must match member for member, with no 'reordering', and no additional members. With EXTENSBILE extensibility, the types can match as long as members

are added only at the end of the existing member list. MUTABLE extensibility allows type matching even if members are rearranged, added and removed. Note that two types of different extensibility cannot 'match' even if they are structurally equivalent. The complete rules for type matching are fairly complex – they are presented in a subsequent section.

Extensibility can also have a significant impact to the usage of the data type. For example, the encode and decode of structures with FINAL extensibility results in the most compact representation on the wire.

NOTE: due to technical issues with the standard specification for EXTENSIBLE, this mode should be avoided. An upcoming version of the standard will introduce a new extensibility called 'APPEND'. This includes an update that removes the issues with 'EXTENSIBLE' while providing the desirable behaviors. EXTENSIBLE will be maintained for backwards compatibility and migration support, but it should be avoided when possible.

2.3.1.1.2 Nested

A structure can be marked as 'nested' by applying the @nested annotation. This indicates that the structure is not to be considered a 'top-level' type, and therefore, the compiler can avoid generating type specific TypeSupport, DataReader, and DataWriter code. This is useful for reducing the volume of generated code.

In IDL:

```
@nested
struct InnerStruct {
     long avalue;
};
```

And in XML:

```
<struct name="InnerStruct"
        nested="true">
  <member name="avalue"
          type="int32"/>
</struct>
```

2.3.1.2 Member Properties

All members of a structure have certain additional properties (in addition to their type and name). These properties can be controlled by means of Annotations. The properties and their effect are described briefly here, further information can be found in the Type Augmentation section.

2.3.1.2.1 Key

Structure types can have a set of members defined to define the 'key'. A unique 'key' value indicates a unique instance in the data model. A member can be indicated as being part of the 'key' by applying the @key attribute. The default behavior for CoreDX DDS is that the 'key' attribute applies recursively to the contents of a 'key' member. That is, if the member is a structure type, and it is marked key, then the members of the embedded structure are all marked key. [This is in contrast to the mechanism specified

by the X-Types standard.] Further information is provided in Section 2.8.7.

2.3.1.2.2 Must Understand

Each member has a 'must_understand' flag. This flag can be set explicitly by use of the @must_understand annotation. In IDL:

```
@extensibility(MUTABLE_EXTENSIBILITY)
struct A {
    @must_understand long a_long;
                     long b_long;
};
```

And in XML:

```
<struct name="A"
        extensibility="mutable">
  <member name="a_long"
          mustUnderstand="true"
          type="int32"/>
  <member name="b_long"
          type="int32"/>
</struct>
```

The 'mustUnderstand' flag indicates if the member must be understood by a receiver. If this flag is false, the receiver is free to ignore this member if it is not known as part of the receiver's data type. If this flag is true, and the receiver does not have a corresponding member in its data type, then the receiver must drop the entire sample of which this member is a part.

The value of a member's "optional" property is unrelated to the value of its "must understand" property. For example, it is legal to define a type in which a non-optional member can be safely skipped or one in which an optional member, if present and not understood, must lead to the entire sample being discarded.

2.3.1.2.3 MemberId

Each member of a structure is assigned a unique ID value. By default this value is a one-up counter, starting at 1. The type designer can manually assign memberId values by applying the @id annotation.

Example IDL:

```
struct A {
    @id(100) long a_long;
};
```

And XML:

```
<struct name="A">
  <member name="a_long"
          id="100"
          type="int32"/>
</struct>
```

It is required that all members of a structure have unique memberId's. The 'memberId' property is meaningful for members of EXTENSIBLE and MUTABLE structure types. It comes into play during

type matching, and may be used in the on-the-wire encoding.

2.3.1.2.4 Optional

Each member has a boolean flag 'optional'. This flag indicates if the member may be omitted. If 'optional' is true, then it is valid for the data type to have no value for this member. [In this case the member is set to NULL.] If a member is 'optional', then it is mapped to a pointer type, much like 'external' members. A member cannot be both 'optional' and 'key'.

2.3.1.2.5 External (aka Shared)

The 'external' property indicates that the member should be mapped to a pointer or reference. This allows the user of the data type to reference data outside of the sample instance. Care must be taken when using 'external' members. The generated 'destructor' code will delete memory referenced by any external members. If the sample is not the owner of that memory, then the user must take care to clear the reference (set to NULL) before calling the destructor code.

2.3.1.3 C Language Mapping

The IDL struct type is mapped to a C struct. Members are listed in the order in which they are presented in the IDL struct. In addition to the struct definition, the code generator emits several methods to operate on instances of the structure, for example 'alloc', 'init', and 'clear' operations.

For example, the following IDL

```
struct A {
    long long_1;
    long long_2;
};
```

maps to:

```
typedef struct A {
   int  long_1;
   int  long_2;
} A;

struct A *     A_alloc ( void );
void           A_free ( struct A * inst );
void           A_init ( struct A * instance );
void           A_clear( struct A * instance );
void           A_copy ( struct A * copy_to, const struct A * copy_from );
```

2.3.1.4 C++ Language Mapping

The C++ mapping is similar to the C, but uses methods. For the IDL “struct A” presented above, the C++ mapping is:

```
  struct COREDX_TS_STRUCT_EXPORT A {
    public:
      /** Constructor, Copy Constructor, Destructor, Assignment operator */
      A();
      A( const A & other );
      ~A();
```

```
      A& operator=( const A & other);

      void init();
      void clear();
      void copy( const A * instance );

      int  get_marshal_size(int offset, int just_keys) const ;
      int  marshal_cdr(unsigned char * buf, int offset, int stream_len, unsigned
char swap, int just_keys) const ;
      int  marshal_key_hash(unsigned char *buf, int offset, int stream_len) const;
      int  unmarshal_cdr(unsigned char * buf, int offset, int stream_len, unsigned
char swap, int just_keys);
      int  unmarshal_key_hash(unsigned char *buf, int offset, int stream_len);

      /* Member vars*/
      int  long_1;
      int  long_2;

    private:

  }; //A
```

2.3.1.5 C# Language Mapping

A structure is mapped to a public class in C#. For example, the above IDL maps to the following C#:

public class A {

```
  // instance variables
  public int long_1;
  public int long_2;

  // …
}
```

2.3.1.6 Java Language Mapping

A structure is mapped to a public class in Java. For example, the above IDL maps to the following Java:

```
public class A {

  // instance variables
  public int long_1;
  public int long_2;

  // ….
};
```

2.3.2 Union Types

Union types are another aggregate type. They consist of a discriminator and a list of potential members. The value of the discriminator determines which one of the potential members are actually present in the union instance. The name of the discriminator is 'discriminator', and that name is

reserved for union types (no other member may be named discriminator).

Each member is associated with one or more values of the discriminator. These values are identified in one of two ways: (1) They may be identified explicitly; it is not allowed for multiple members to explicitly identify the same discriminator value; and, (2) at most one member of the union may be identified as the “default” member; any discriminator value that does not explicitly identify another member is considered to identify the default member. These two mechanisms together guarantee that any given discriminator value identifies at most one member of the union. (Note that it is not required for every potential discriminator value to be associated with a member.)

The mapping from discriminator value to member is defined by the union type and does not differ from instance to instance.

The value of the member associated with the current value of the discriminator is the only member value considered to exist in a given object of a union type at a given moment in time. However, the value of the discriminator field may change over the lifetime of a given object, thereby changing which union member’s value is observed. It is not defined whether, upon switching from a discriminator value x to a different value y and then immediately back to x, the previous value of the x member will be preserved.

Example IDL union definition:

```
union U1 switch(octet) {
  case 0: long zero_long;
  case 2: octet two_byte;
  default: string default_string;
};
```

And in XML:

```
<union name="U1">
  <discriminator
    type="byte"/>
  <case>
    <caseDiscriminator value="0" />
    <member name="zero_long"
            id="1"
            type="int32"/>
  </case>
  <case>
    <caseDiscriminator value="2" />
    <member name="two_byte"
            id="2"
            type="byte"/>
  </case>
  <case>
    <caseDiscriminator value="default" />
    <member name="default_string"
            id="3"
            type="string"/>
  </case>
</union>
```

2.3.2.1 C Language Mapping

This IDL union definition

```
union U switch(boolean) {
  case TRUE: long true_long;
  case FALSE: octet false_byte;
};
```

maps to the following:

```
  typedef struct U {
      unsigned char  discriminator;
      unsigned char  _initialized;
      union {
         int  true_long;
         unsigned char  false_byte;
      } _u;
  } U;

struct U      *U_alloc ( void );
void           U_free ( struct U * inst );
void           U_init ( struct U * instance );
void           U_clear( struct U * instance );
void           U_copy ( struct U * copy_to, const struct U * copy_from );
```

2.3.2.2 C++ Language Mapping

The above union IDL ('U') maps to the following C++ code:

```
  class U {  //
    public:
      unsigned char _discriminator;
      unsigned char  _initialized;
      union {
         int  _pd_true_long;
         unsigned char  _pd_false_byte;
      } _u;
    public:
      // Constructor, Copy Constructor, Destructor, Assignment operator
      U();
      U( const U & other );
      ~U();
      U& operator=( const U & other);
      void      discriminator(unsigned char d)
            { _discriminator = d; _initialized = 1; }
      unsigned char  discriminator() const
            { return _discriminator; }

      void init();
      void clear();
      void copy( const U * instance );

      int  get_marshal_size(int offset, int just_keys) const ;
      int  marshal_cdr(unsigned char * buf, int offset, int stream_len,
                       unsigned char swap, int just_keys) const ;
      int  marshal_key_hash(unsigned char *buf, int offset, int stream_len) const;
      int  unmarshal_cdr(unsigned char * buf, int offset, int stream_len,
                       unsigned char swap, int just_keys);
```

```
    int  unmarshal_key_hash(unsigned char *buf, int offset, int stream_len);

    /* Member vars*/
    int true_long() const { return _u._pd_true_long; }
    void true_long(  int _v) {
      clear();
      _u._pd_true_long = _v;
      this->discriminator(DDS_TRUE);
    }
    unsigned char false_byte() const { return _u._pd_false_byte; }
    void false_byte(  unsigned char _v) {
      clear();
      _u._pd_false_byte = _v;
      this->discriminator(DDS_FALSE);
    }
  };
```

NOTE:

- If the union type includes any 'constructed' members, they are not included in the internal union '_u'; rather, they are promoted to be top-level members of the generated C++ class. That is, only 'primitive' data types can be put into the internal union '_u'.
- We generate an accessor and modifier (or, multiple modifiers depending on the member type) for each 'case' in the union type.
- There are accessor and modifier members for the discriminator field.

2.3.2.3 C# Language Mapping

The above union IDL ('U') maps to the following C# code:

```
public class U :  DdsType  {
  public U() {
    __init = false;
  }

  public U init() {
    __init = true;
    __disc = (bool)false;
    return this;
  }

  public void clear() {
    __init = false;
    // Skipping non-dynamic symbol: true_long
    // Skipping non-dynamic symbol: false_byte
  }

  public void copy( Object f ) {
    U from = (U)f;
    __init = from.__init;
    __disc = from.__disc;
    if ((discriminator()==true)) {
```

```
            this.true_long = from.true_long;
        }
      if ((discriminator()==false)) {
            this.false_byte = from.false_byte;
        }
    }

    public bool discriminator() {
      return __disc;
    }

    // true_long property
    public  int true_long {
       get {
         if (__init==false) throw new System.ArgumentException();
         if ((__disc==true))
               return __true_long;
          throw new System.ArgumentException();
       }
       set {
          __init = true;
          __disc = (bool) true;
          __true_long = value;
       }
    }
    // false_byte property
    public  byte false_byte {
       get {
         if (__init==false) throw new System.ArgumentException();
         if ((__disc==false))
               return __false_byte;
          throw new System.ArgumentException();
       }
       set {
          __init = true;
          __disc = (bool) false;
          __false_byte = value;
       }
    }
    public  bool   __disc;
    public  bool __init;
    private  int  __true_long;
    private  byte  __false_byte;
}; // U
```

2.3.2.4 Java Language Mapping

The above union IDL ('U') maps to the following Java code:

```
final public class U {

  public U() {
    __init = false;
  }

  public U init() {
    __init = true;
    __disc = false;
```

```
    return this;
  }

  public void clear() {
    __init = false;
  }

  public void copy( U from ) {
    __init = from.__init;
    __disc = from.__disc;
    this.true_long = from.true_long;
    this.false_byte = from.false_byte;
  }

  public boolean discriminator() {
    return __disc;
  }

  // true_long accessor
  public  int true_long() throws Exception {
   if (__init==false) throw new Exception();
   if ((__disc==true))
         return true_long;
    // TODO: throw BAD_OPERATION exception ?
    throw new Exception();
  }
  // true_long modifier
  public void true_long( int __val) {
    __init = true;
    __disc = true;
    true_long = __val;
  }
  // false_byte accessor
  public  byte false_byte() throws Exception {
   if (__init==false) throw new Exception();
   if ((__disc==false))
         return false_byte;
    // TODO: throw BAD_OPERATION exception ?
    throw new Exception();
  }
  // false_byte modifier
  public void false_byte( byte __val) {
    __init = true;
    __disc = false;
    false_byte = __val;
  }
  public  boolean __disc;
  public  boolean __init;
  public  int true_long;
  public  byte false_byte;
}; // U
```

2.4 Type Aliases

The IDL 'typedef' provides an alternate name for an already-existing type. The alternate name can be helpful for suggesting particular uses and semantics to human readers, making it easier to repeat complex type names for human writers, and simplifying certain language bindings. An alias/typedef

does not introduce a distinct type; it provides an alternative name by which to refer to a type.

IDL alias:

```
typedef long MyLong;
```

XML alias:

```
<typedef name="MyLong"
         type="int32" />
```

2.4.1.1 C Language Mapping

IDL 'typedef' maps to typedef in C.

```
  typedef int MyLong;
```

2.4.1.2 C++ Language Mapping

IDL 'typedef' maps to typedef in C++.

```
  typedef int MyLong;
```

2.4.1.3 C# Language Mapping

The C# language mapping of an alias simply replaces the alias with the fundamental type indicated by the alias. Any number of nested aliases will be reduced to the fundamental type. The typedef does not generate any C# code.

2.4.1.4 Java Language Mapping

The Java language mapping of an alias simply replaces the alias with the fundamental type indicated by the alias. Any number of nested aliases will be reduced to the fundamental type. The typedef does not generate any Java code.

2.5 Constants

IDL constant:

```
      const long A_LONG_CONST = 100;
```

XML constant:

```
      <const name="A_LONG_CONST"
             type="int32"
             value="100" />
```

2.5.1.1 C Language Mapping

```
      #define A_LONG_CONST (100)
```

2.5.1.2 C++ Language Mapping

```
      static const int A_LONG_CONST = (100);
```

2.5.1.3 C# Language Mapping

```
      namespace A
      {
```

```
  public class A_LONG_VALUE {
    public const int value = (int)(100);
  }
}
```

2.5.1.4 Java Language Mapping

```
package A;

public interface A_LONG_VALUE {
  public static final int value = (int)(100);
}
```

2.6 Annotations

IDL Annotations are a means of augmenting the type information in the IDL input. Annotations are applied to a type by specifiying the annotation name (with an ampersand '@' symbol prefix) including scope, if relevant or desired, and listing values for any annotation parameters. The annotation syntax and the pre-defined annotation instances are discussed in detail in the Type Augmentation section.

The 'builtin' annotations are used to control the mapping of types into a language specific representation or to tailor the behavior of the generated type support code.

Additional, user defined, annotation types can be added by specifying a new annotation name and members with the IDL annotation declaration syntax.

```
annotation @MyAnnotation {
   bool isGood;
};
```

Once an annotation type has been defined, it can be applied in the same manner as any other built-in annotation. However, because the IDL parser does not have any a-prior knowledge about the user-defined annotation, it does not impact code generation.

2.7 Interfaces

The RPC over DDS implementation makes use of 'interfaces' to define the data types and operations used by the RPC API. In IDL the interface construct looks like this:

```
@nested
exception TooBig {};

@service
interface Foo {
  long    op1( long param );    // operation taking one param, returning a long
  long    op2 ( );              // taking no parameters, returning a long
  void    op3 ( long val ) raises TooBig; // returns void, may raise exception
};
```

The above example declares an interface named Foo with three operations. The '@service' annotation indicates that this interface defines an RPC over DDS interface. The exception 'TooBig' is used by one of the operations. For more information on RPC over DDS, see the **CoreDX RPC Programmers**

Guide.

2.8 Type Augmentation

A type definition can be augmented by attaching additional information. For example, the DDS idea of 'key' members in a structure or union. In IDL, this augmentation is accomplished via an 'annotation' construct. In XML, the augmentation is accomplished by additional properties on an element (with the exception of 'Verbatim', which is a separate element). In general, we refer to this type augmentation as an '**annotation**', regardless of the actual mechanism used to attach the extra information. Several types of annotations are defined in the standard. These annotations include:

bit_bound
bitmask
extensibility
external [aka: shared]
id
key
must_understand
nested
optional
verbatim

Some annotations can be applied to type definitions (for example, *extensibility* or *nested*), while others apply only to members of a type (for example, *key* or *id*). The allowed usage of each annotation is described below.

2.8.1 Annotation Syntax

In IDL, annotations are indicated with an '@' prefix. Some annotations take no values, their meaning is conveyed simply by their presence (for example @key); while others take one or more values to completely define their meaning. Values can be provided to an annotation by providing a list of parameters of the form 'name=value'. For example:

```
@my_annotation(id=5, color=”green”)
```

If an annotation has one parameter, and that parameter is named 'value', then a shorthand syntax is allowed:

```
@an_annotation(5) is equivalent to @an_annotation(value=5)
```

If an annotation application omits values for some (or all) of the annotation's parameters, then those parameters are initialized with their default value. [The default value can be specified in the definition of the annotation.] If there is no default value specified for a parameter, then the default is taken as zero or an empty string.

In most cases an annotation can be applied in 'prefix' style or 'postfix' style. Further, prefix and postfix applications can be mixed. For example:

Prefix:

```
@key  long my_key;
```

Postfix:

```
long my_key;  //@key
```

2.8.2 Built-in Annotations

The following sections describe the meaning and use of each of the annotations.

2.8.3 Bit Bound

The *@bit_bound* annotation is used to specify a number of bits. It is applicable to enum types and bitmask types. For enumerations, it can take a value between 1 and 32 inclusive, and the default is 32. For bitmask types, its range is extended to 1 .. 64 inclusive, with a default of 32.

For example:

```
@bit_bound(8) enum SmallEnum { VAL1, VAL2 };
```

This has the effect of declaring an enumeration type 'SmallEnum' that is represented as an 8bit value. Note, the bit_bound is rounded up to the nearest natural machine data size (that is: 8bits, 16bits, 32bits, or 64bits).

2.8.4 BitMask

This annotation is used to 'convert' and enumeration type into a Bit Set. This annotation is useful in those cases where the IDL must be acceptable by a parser that does not understand the 'bitmask' keyword. However, if the compiler doesn't process the @bitmask annotation, then the type will be considered a basic enumeration, and the named constant values will not be computed the same as a compiler that understands @bitmask. This could lead to errors, perhaps undetectable until run-time. For this reason, the use of the @bitmask annotation is discouraged. Instead, it is better to use the first-class 'bitmask' type, and be alerted to the incompatibility at code generation time.

2.8.5 Extensibility

Extensibility refers to the ability to change or extend a data type. This has significant implications to the algorithm that determines if two data types 'match' or are 'compatible'. In previous versions of CoreDX DDS, two types were considered to be compatible if the structure of the types matched exactly including things such as keys, string lengths, and array sizes. The X-Types standard introduces more complex rules for type compatibility that include the ability to add members to a data type or re-order members. In this case, two types can be compatible without exactly matching structurally. There are three types of 'extensibility' defined in the standard: Final, Extensible, and Mutable.

The 'Final' extensibility essentially matches the behavior of 'pre X-Types' DDS systems. Data types that are marked as 'Final' are not compatible with other types unless they match structurally.

A type marked 'Extensible' is compatible with another type if the other type is a strict super set of the original type, and the members in common are also declared in the same order and position.

A type marked 'Mutable' is the most flexible. The members of two types need not overlap completely, and structure members may be declared in different orders and positions. As long as there exists a set of members in common between the two types, and each member in that set has the same 'ID', and name in both types [and the 'key' fields match exactly], then the two types are considered compatible.

These rules are slightly complicated by the 'must_understand' attribute (described below). If a member in type T1 is marked as 'must_understand', but is not present in a type T2, then the type T1 is not assignable to T2 (which means that the types are incompatible).

The rules for type compatibility are discussed further inSection 3 Type Discovery and Type Matching.

2.8.6 ID

Every member in a structure or union type is assigned a 'member id' (or ID). This ID is used (in the case of Extensible or Mutable extensibility) to determine type matching, and is potentially included in the encoded data to facilitate decoding. The @id annotation allows the user to control how ID's are assigned to type members. Without the annotation, member id is assigned as a one-up counter, starting at zero, proceeding from the largest value seen so far in processing the type.

For example:

```
struct A {
    long a1;  // will be assigned member_id = 0
    long a2;  // will be assigned member_id = 1
};

struct B {
    @id(10) long b1;  // will be assigned member_id = 10
            long b2;  // will be assigned member_id = 11
};

struct C {
    @id(10) long c1;  // will be assigned member_id = 10
    @id(1)  long c2;  // will be assigned member_id = 1
            long c2;  // will be assigned member_id = 11
};
```

NOTE: It is expected that a future update to the X-Types standard will define a new algorithm for automatic ID value generation (rather than the current 1-up counter mechanism). The ID will likely be based on a hash of the member name. This will enable ID values to remain the same, even if members are rearranged in the data type. This may have an impact if you develop types and rely on the 'auto-assignment' feature. A future version of the compiler will include an option to select between the current one-up counter and any other algorithm (for example, hash based).

2.8.7 Key

The @key annotation is used to mark those members that make up the key of the data type. In general, any member of a structure can be marked as a key, and a union discriminator can be marked as key. A 'key' member cannot also be marked 'optional' (see Optional, below).

In CoreDX DDS the key of a data type is determined recursively. For example

```
struct B {
      @key long  b1;
           long  b2;
};

struct A {
      @key long  a1;
           B     b;
};
```

The complete key of 'A' is (A.a1, A.b.b1). This behavior is consistent with the key definition logic used in previous versions of CoreDX DDS. However, the X-Types standard defines a slightly different approach, and the key of 'A' would be just (A.a1). In order to get the key fields from 'B' included in type 'A', one would have to prefix B with an @key annotation, like this:

```
struct B {
      @key long  b1;
           long  b2;
};

struct A {
      @key long   a1;
      @key B      b;
};
```

In this case, with strict X-Types key behavior, the key would be (A.a1, A.b.b1). With CoreDX DDS key behavior, the key would be (A.a1, A.b.b1, A.b.b2).

2.8.8 Must Understand

The @must_understand annotation can be applied to member(s) of a structure or union type. This indicates that the receiver of this data type must be able to understand (parse) the data for this member. If the receiver type does not contain a matching definition for the 'must understand' member, then the receiver may fail to parse data that includes the member. For this reason, two types are considered incompatible if the receiver type omits one or more 'must understand' members.

2.8.9 Nested

The @nested annotation can be applied to an aggregate data type (structure or union) and it indicates that the type does not require full code generation. For example, a nested type 'Foo' will not generate FooDataReader, FooDataWriter, and FooTypeSupport code. This can be helpful to reduce the volume of generated code.

2.8.10 Optional

The @optional annotation can be applied to a member of an aggregate type (structure or union). This indicates that the member may in some cases be absent. An absent member is indicated by a 'null' value for the member. [This implies that the member is mapped to a 'pointer' or 'reference' type in the language mapping.]

A 'key' member cannot be 'optional'.

2.8.11 External [aka: Shared]

The @external annotation is applicable to a member of an aggregate type (structure or union). It can also be applied to array and sequence elements. This annotation indicates that the member is to be mapped to a 'pointer' or 'reference' type in the language binding. It has no other impact on the data type. In the CoreDX DDS implementation, an external member is initialized to NULL by a data types constructor. If the member is non-NULL when the destructor is called, the destructor will attempt to recursively destroy the member. The application is responsible for setting the member pointer, and removing it, as appropriate.

2.8.12 Verbatim

The @verbatim annotation allows a type designer to insert text into the generated code. This annotation can be applied to any type definition. This annotation has several properties that can be utilized to control when and where the text is inserted.

language: this parameter is a string that specifies which output language this verbatim text should be output for. The default value of this parameter is “*” which will match any output language. The other values that have meaning for CoreDX DDS are “c”, “c++”, “java”, and “csharp”.

placement: The placement parameter defines where the verbatim text should be inserted in the output code. The default value is “before-declaration”. The possible values are:

Verbatim Placement	Meaning
begin-declaration-file	The text is inserted at the beginning of the file containing the declaration of the associated type before any type declarations.
before-declaration	The text is inserted immediately before the declaration of the associated type.
begin-declaration	The text is inserted into the body of the declaration of the associated type before any members or constants.
end-declaration	The text is inserted into the body of the declaration of the associated type after all members or constants.
after-declaration	The text is inserted immediately after the declaration of the associated type.
end-declaration-file	The text is inserted at the end of the file containing the declaration of the associated type after all type declarations.

Table 3: Verbatim Placement values

text: this string parameter is copied directly into the generated code subject to the 'language' and 'placement' parameters.

3 Type Discovery and Type Matching

CoreDX DDS exchanges type information during the entity discovery process. That is, when announcing the existence of the Reader or Writer entity, the data type information is included with the entity's QoS policies. This allows the peers to perform a 'type compatibility' test before matching two entities. [Note, this behavior was supported by CoreDX DDS prior to X-Types; but it used a non-standard extension to the discovery information.]

The rules for determining type compatibility are non-trivial. With the addition of 'Extensible' and 'Mutable' data types, simple structural equivalence is no longer sufficient to determine type compatibility.

The rules for type compatibility are based on the 'is-assignable-from' predicate. This is necessary to capture the behavior of a 'sample' produced by a Writer that must be consumed by a Reader. The type of the written sample must be assignable to an instance of the type known by the Reader. [It may not be true that the reverse assignability is possible.]

The rules for the 'is-assignable-from' predicate are described in detail in the following sections. We consider T1 is-assignable-from T2.

3.1 Primitive Types

Any primitive type is assignable to that exact primitive type. No type 'coercion' is allowed as the wire representation is likely different; for example, a long (4 bytes) vs a short (2 bytes). Allowing type compatibility between long and short would force the reader to recognize the different size of the type in its encoded form – adding undesired overhead to the encoding and processing.

3.2 Collection Types

Collection types include **string**, **array**, **sequence**, and **map**. The collection is assignable if the collection elements are assignable and if the collection bound is compatible. For strings, the elements are either char8 or char32. For strings, maps, and sequences the bound is compatible if T1.bound >= T2.bound. [That is, if the published collection count will always be smaller or equal to the subscribed count.] For arrays, the bound is compatible only if T1.size == T2.size. For a map, the 'key type' is tested for is-assignable-from in addition to testing the 'value type'.

3.3 BitMask Types

BitMask T1 is-assignable-from BitMask T2 if T1.bound == T2.bound. Further, a BitMask can be assigned from an 'integral' type if the integer type has a size that matches the underlying type of the BitMask. That is:

0 < T1.bound <= 8 is-assignable-from UINT8
8 < T1.bound <= 16 is-assignable-from UINT16
16 < T1.bound <= 32 is-assignable-from UINT32
32 < T1.bound <= 64 is-assignable-from UINT64

Table 4: BitMask bitbound sizes

3.4 Enumeration Types

Enumeration T1 is-assignable-from Enumeration T2 if and only if:

1) Any constants that have the same name in T1 and T2 also have the same value, and any constants that have the same value in T1 and T2 also have the same name; and

2) T1.extensibility == T2.extensibility; and

3) if (T1.extensibility == Extensible) then, the following is true:

a) for each constant index ‘i’ in T1 the constant in T1 at that index c1[i] and the constant in T2 at that index c2[i], if c2[i] exists, have the same name.

4) if (T1.extensibility == Final) then, the following are also true:

a) The number of constants in T1 is equal to the number of constants in T2; and

b) For each constant index ‘i’ in T1 the constant in T1 at that index c1[i] and the constant in T2 at that index c2[i] have the same name.

3.5 Aggregation Types

For aggregation types, is-assignable-from is based on the extensibility of the type and the is-assignable-from predicate of the types’ members. The correspondence between members in the two types is established based on their respective member IDs and on their respective member names.

3.5.1 Structure Types

For the purposes of determining 'is-assignable-from' for structure types, members belonging to base types of T1 or T2 shall be considered “expanded” inside T1 or T2 respectively, as if they had been directly defined as part of the sub-type.

Structure type T1 is-assignable from Structure type T2 if and only if the following holds true:

1) T1.extensibility == T2.extensibility; and

2) T1.keys.count == T2.keys.count (that is, they have the same number of key members); and

3) For each member “m1” that forms part of the key of T1 (directly or indirectly), there is a corresponding member “m2” that forms part of the key of T2 (directly or indirectly) with the same member id (m1.id == m2.id) where m1.type is-assignable-from m2.type; and

4) Any members in T1 and T2 that have the same name also have the same ID and any members with the same ID also have the same name; and

5) For each member “m1” in T1, if there is a member m2 in T2 with the same member ID then m1.type is-assignable-from m2.type; and

6) For each member “m2” in T2 for which both **optional** is false and **must_understand** is true there is a corresponding member “m1” in T1 with the same member ID; and

7) There is at least one member “m1” of T1 and one corresponding member “m2” of T2 such that m1.id == m2.id (that is, the two type must share at least one member); and

8) if (T1.extensibility == Extensible) then, the following is true:

a) for each member index ‘i’ in T1 the member in T1 at that index m1[i] and the member in T2 at that index m2[i], if m2[i] exists, have the same member ID and the same value of the ‘optional’ attribute and m1[i].type is strongly assignable from m2[i].type.

9) if (T1.extensibility == Final) then, the following is true:

a) The number of members in T1 is equal to the number of members in T2; and

b) For each member index ‘i’ in T1 the member in T1 at that index m1[i] and the member in T2 at that index m2[i] have the same member ID and the same value of the ‘optional’ attribute and m1[i].type is strongly assignable from m2[i].type.

NOTE: prior to the implementation of X-Types, DDS data types essentially behaved as FINAL extensibility. In order to achieve compatibility with non-X-Types deployed components (that is components deployed with a DDS implementation that does not support X-Types), it is necessary to mark types as FINAL.

3.5.2 Union Types

In general, Union type T1 is-assignable-from Union type T2 if and only if it is possible to identify the appropriate T1 member based on the T2 discriminator value and the is-assignable-from predicate holds for both the discriminator and the selected member. More specifically, T1 is-assignable-from T2, if and only if:

1) T1.extensibility == T2.extensibility; and

2) T1.discriminator.id == T2.discriminator.id and T1.discriminator.type is-assignable-from T2.discriminator.type; and

3) Either the discriminators of both T1 and T2 are keys or neither are keys; and

4) Any members in T1 and T2 that have the same name also have the same ID and any members with the same ID also have the same name; and

5) For each member “m1” in T1, if there is a member m2 in T2 with the same member ID then m1.type is-assignable-from m2.type if T1 is mutable or strongly assignable if T1 is final or extensible; and

6) A discriminator value appearing in a non-default label of T2 selects a member m2. If the same discriminator value selects a member m1 of T1, then m1.id == m2.id; and

7) A discriminator value appearing in a non-default label of T1 selects a member m1. If the same discriminator value selects a member m2 of T2, then m1.id == m2.id; and

8) If both T1 and T2 have a default label, then the IDs of the members selected by those labels are equal; and

9) if (T1.extensibility == Final) then, the number of members in T1 is equal to the number of members in T2.

4 Dynamic Types and Dynamic Data

The CoreDX DDS type system includes the option to define data types in code 'on-the-fly'. This can be used to develop dynamic general-purpose analysis tools that can discover types at run-time and interact with entities using those types. Further, it can be helpful when the volume of code generated by IDL is undesirable, or any other time when an application needs to construct a type at run-time.

The DynamicType API includes the DynamicType and DynamicData entities, and the associated TypeSupport, DataReader, and DataWriter instances that work on DynamicData. A DynamicData object represents an instance of a particular DynamicType.

A DynamicType instance represents a specific data type. It supports the complete type system that is available through IDL and XML; in other words, any DDS type that is legal in IDL can also be represented by a DynamicType. For example, it supports all primitive types, all collection and aggregate types, and all type enhancements (key, external, optional, etc). A DynamicType can be built manually by calling on the API for DynamicTypeBuilderFactory and DynamicTypeBuilder. Alternatively, a DynamicType can be constructed from a TypeObject instance; for example, a type learned through discovery.

A DynamicTypeSupport can be created from a DynamicType, and then can be used by a DynamicDataReader or DynamicDataWriter.

The DynamicData API provides methods to access or modify data elements by name or id, and exposes the structure of the data so that an application can traverse complex embedded data instances.

4.1 API

4.1.1 DynamicTypeBuilderFactory

The DynamicTypeBuilderFactory is a singleton object that is used to create and destroy DynamicTypeBuilder instances. Further, it includes methods to create a DynamicTypeBuilder from an XML document or a URI pointing to an XML document. [Note, this feature is not yet implemented in CoreDX DDS.]

4.1.2 DynamicTypeBuilder

A DynamicTypeBuilder object represents the state of a particular type defined according to the Type System. It is used to instantiate concrete DynamicType objects. It includes methods to access or modify members (if representing a collection or aggregate type) by name or by id. The 'build()' method constructs a DynamicType instance based on the current state of the DynamicTypeBuilder. Subsequent changes to the DynamicTypeBuilder will not impact any current DynamicType instances.

A DynamicTypeBuilder is obtained from the DynamicTypeBuilderFactory, and should be destroyed by calling the DynamicTypeBuilderFactory::delete_type_builder() method.

4.1.3 DynamicType

An instance of DynamicType represent a type's schema: its physical name, kind, member definitions (if any), and so on. A DynamicType can be passed as a parameter to the static method DynamicTypeSupport::create_type_support() to construct a DynamicTypeSupport.

A DynamicType is obtained via the DynamicTypeBuilderFactory::get_primitive_type() method, or from a DynamicTypeBuilder::create_type() method. All DynamicType instances should be destroyed by calling the DynamicTypeBuilderFactory::delete_type() method.

4.1.4 DynamicTypeMember

A DynamicTypeMember represents a "member" of a type. A "member" in this sense may be a member of an aggregated type, a constant within an enumeration, or some other type substructure. A DynamicTypeMember contains a MemberDescriptor, a set of flags, and a set of annotations (maybe empty).

4.1.5 TypeDescriptor

A TypeDescriptor comprises the state of a type; that is, the 'kind', the name, the 'base_type' if it is an alias or if it is a structure that derives from some other type, the discriminator type if it is a union, the bound, if it is a collection, enum or bitmask, the element_type if it is a collection, and the key_element_type if it is a map.

4.1.6 MemberDescriptor

A MemberDescriptor represents the state of a DynamicTypeMember. It contains the member name, the member id, the DynamicType that represents the member type, the index, the default_value, and the sequence of labels and a flag indicating if it is the default case (if part of a union).

4.1.7 DynamicDataFactory

The DynamicDataFactory is a singleton object responsible for creating DynamicData instances. It provides two methods: create_data(DynamicType) and the corresponding delete_data(DynamicData).

4.1.8 DynamicData

A DynamicData object represents an individual data sample. It provides reflective getters and setters for the members of that sample. Many of the properties and operations on DynamicData refer to values

within the sample, which are identified by name, member ID, or index. What constitutes a value within a sample, and which means of accessing it are valid, depends on the type of the sample.

- If this instance is of an aggregated type, values correspond to the type's members and can be accessed by name, member ID, or index.
- If this instance is of a sequence or string type, values correspond to the elements of the collection. These elements must be accessed by index; the mapping from index to member ID is unspecified.
- If this object is of a map type, values correspond to the values of the map. Map keys are implicitly converted to strings and can thus be used to look up map values by name. Map values can also be accessed by index, although the order is unspecified.
- If the object is of an array type, values correspond to the elements of the array. These elements must be accessed by index; the mapping from index to member ID is unspecified. If the array is multi-dimensional, elements are accessed as if they were "flattened" into a single-dimensional array in the order specified by the IDL specification.
- If the object is of a bit set type, values correspond to the flags within the bit set and are all of Boolean type. Named flags can be accessed using that name; any bit within the bound of the bit set may be accessed by its index. The mappings from name and index to member ID are unspecified.
- If the object is of an enumeration or primitive type, it has no contained values. However, the value of the sample itself may be indicated by "name" using a nil or empty string, by "ID" by passing MEMBER_ID_INVALID, or by "index" by passing index 0.

Note that indices used here are always relative to other values in a particular DynamicData object. Even though member definitions within aggregate types have a well-defined order, the same is not true within data samples or across data samples.

Specifically, the index at which a member of an aggregated type appears in a particular data sample may not match that in which it appears in the corresponding type and may not match the index at which it appears in a different data sample.

There are several reasons for these inconsistencies:

- The producer of the sample may be using a slightly different variant of the type than the consumer, which may add to, or omit elements from, the set of members known to the consumer.
- An optional member may have no value; in such a case, it will be omitted, thereby decreasing the index of every subsequent member.
- A non-optional member may likewise be omitted (which semantically is equivalent to it taking its default value). An implementation may discretionarily omit such members (e.g., to save

space).

- Preserving member order is not necessary or even desirable (e.g., for performance reasons) for certain data representations.

The DynamicData API provides methods to get or modify values within the data instance. The naming scheme of the accessors and modifiers is intuitive, for example get_int32_value() and set_int32_value().

DynamicData instances obtained by calling DynamicDataFactory::create_data() should be destroyed by calling DynamicDataFactory::delete_data().

4.1.9 DynamicDataReader

The DynamicDataReader provides a DataReader interface that is tailored to support reading a specific DynamicType. As with any TypeSupport, the corresponding DynamicTypeSupport must have been registered previously with the DomainParticipant. The DynamicDataReader access data samples of type DynamicData that have a type construct identified by the DynamicType.

4.1.10 DynamicDataWriter

The DynamicDataWriter provides a DataWriter interface that is tailored to support writing samples of a specific DynamicType. The specific DynamicTypeSupport must have been registered previously with the DomainParticipant. The writer accepts DynamicData samples with a construct that is identified by the specific DynamicType.

4.2 Usage / Examples

4.2.1 C

The C API to DynamicTypes and DynamicData is presented in the dds/xtypes.h header file.

```
#include <dds/xtypes.h>
```

The implementation is provided in the dds_dyntype library

Operating System	static library	dynamic library	dependencies
linux	libdds_dyntype.a	libdds_dyntype.so	libdds.a / libdds.so
Windows	dds_dyntype_static.lib	dds_dyntype.dll	dds_static.lib / dds.dll

Table 5: C DynamicType libraries

Here is a snipet of code that creates a structure with a single member of type int32 named 'an_int32'.

```
  #include <dds/xtypes.h>

...

  DDS_ReturnCode_t ddsret;
  DDS_DynamicTypeBuilderFactory  dtbf;
  DDS_DynamicTypeBuilder         dstruct;
```

```
dtbf    = DDS_DynamicTypeBuilderFactory_get_instance();
dstruct = DDS_DynamicTypeBuilderFactory_create_structure_type( dtbf );
if (dstruct != NULL)
  {
    DDS_MemberDescriptor  md;
    DDS_MemberDescriptor_init(&md);
    strcpy(md.name, "an_int32");
    md.id            = 0;
    md.type          = DDS_DynamicTypeBuilderFactory_get_primitive_type(dtbf, DDS_INT_32_TYPE);
    md.index         = 0; /* this defines the order of members expected in a CDR serialized stream */

    ddsret = DDS_DynamicTypeBuilder_add_member(dstruct, &md);
    if (ddsret != DDS_RETCODE_OK)
      error...;
    DDS_MemberDescriptor_clear(&md); /* add_member copies the descriptor */

    ...
  }
```

4.2.2 C++

The X-Types DynamicType API is presented in the C++ include file dds/xtypes.hh.

```
#include <dds/xtypes.hh>
```

The C++ DynamicType implementation is provided in the dds_cpp_dyntype library

Operating System	static library	dynamic library	dependencies
linux	libdds_cpp_dyntype.a	libdds_cpp_dyntype.so	libdds_cpp.a + libdds.a / libdds_cpp.so + libdds.so
Windows	dds_cpp_dyntype_static.lib	dds_cpp_dyntype.dll	dds_cpp_static.a + dds_static.lib / dds_cpp.dll + dds.dll

Table 6: C++ DynamicType libraries

4.2.3 C#

The C# DynamicType API is not yet available.

4.2.4 Java

The X-Types DynamicType API is included in the standard **coredx_dds.jar**, so no additional JAR files are required.

5 CoreDX DDL Compiler

5.1 Command-line Options

-a	Align cdr data with respect to CDR encapsulation header (backards compatible to CoreDX v3.2 and older)
-b <filename prefix>	Filename prefix (applies to C and C++)
-c	Generate extra content filter helper routines
-d <destination directory>	Specify the directory for all output files
-D <define>	Define a preprocessor macro (passed to preprocessor)
-e <b|l>	Specify the endian type for target platform (big | little). Default is the endian of the current host.
-E <e|f|m>	Specify the default extensibility for aggregate types (extensible, final, or mutable)
-f <input filename>	Specify the input file
-F	Support full X-Types type system (without this, types are fully backwards compatible)
-g	Guard macros should use full path (C and C++)
-G <guard variable>	Specify the guard macro name (C and C++)
-i [^][gpOsTxX]	include (or exclude with ^ prefix) code generation items: g : generate Foo::get_field() (content filter support) O: generate TypeObject data in TypeSupport p : generate Foo::print() routine s : generate extra data validation in unmarshal code T : generate TypeCode data in TypeSupport x : generate extra typedefs X : generate extra X-Types defined API's (get_type, create_sample(), create_dynamic_sample())
-I <include path>	Specify include path (passed to preprocessor)
-l <c|cpp|csharp|java>	Specify the output language
-L <license path>	Specify the path to the coredx license file
-p <preprocessor>	Specify the preprocessor to use (default: coredx_cpp)
-s	Don't generate code for 'included' files
-S	Strip path from generated #include statements (only relevant if '-s' is in effect)
-t [gpOsTxX]	Toggle code generation items (see -i option)
-v	Print the version of the coredx compiler
-X	Specify that input file is in XML format

Table 7: CoreDX DDS IDL compiler command-line (coredx_ddl)

6 EXAMPLES / PATTERNS

6.1 Type Truncation

This can arise when the writer is publishing a type (TW) that contains “more data” than the reader type (TR) expects. In order for, the Reader and Writer to match, the data types must be determined to be compatible. [More specifically TW must be assignable to TR.]

The assignability rules vary based on the extensibility of the types. [NOTE: for a type to be assignable to another type, the extensibility of the two types must match.]

For example, consider the following two structures:

```
struct WriterType {
      long  x;
      long  y;
      long  z;
};
```

and,

```
struct ReaderType {
      long  x;
      long  y;
};
```

If the extensibility of the types is FINAL, then they are not assignable because the number of members does not match. As a result, the Reader and Writer would not match.

If the extensibility is EXTENSIBLE or MUTABLE, then the types are assignable.

When the reader receives a sample, the “extra” data (member 'z' in this case) will be quietly discarded, and the application will be presented with a sample containing 'x' and 'y'.

In order to prohibit the data truncation, the writer type could be defined with the @must_understand annotation assigned to the member 'z' (or, for completeness, to all members). For example:

```
struct WriterType {
      long  x;
      long  y;
      @must_understand long  z;
};
```

Then, a reader would be obligated to accept all of the data or none. As a result, the absence of member 'z' in the ReaderType would cause the types to not be assignable, and the Reader and Writer would not match.

Consider the more complex example:

```
struct W_InnerType{
      long   x;
      long  y;
      long  z;
```

```
};

struct WriterType {
      long          kind;
      W_InnerType   loc;
      long          extra;
};
```

and,

```
struct R_InnerType{
      long   x;
      long  y;
};

struct ReaderType {
      long        kind;
      R_InnerType loc;
};
```

If extensibility is FINAL or EXTENSIBLE, the types are not assignable.

If Extensibility is MUTABLE, then the types are assignable, and the reader will truncate the data as expected.

[NOTE: The behavior of this example depends on the X-Types version, the X-Types v1.1 specification (expected late 2017), will specify slightly different behavior.]

6.2 Type Expansion

This can arise when the writer is publishing a type (TW) that contains “less data” than the reader type (TR) expects. In order for, the Reader and Writer to match, the data types must be determined to be compatible. [More specifically TW must be assignable to TR.]

The assignability rules vary based on the extensibility of the types. [NOTE: for a type to be assignable to another type, the extensibility of the two types must match.]

For example, consider the following two structures:

```
struct WriterType {
      long  x;
      long  y;
};
```

and,

```
struct ReaderType {
      long  x;
      long  y;
      long  z;
};
```

If the extensibility of the types is FINAL, then they are not assignable because the number of members does not match.

If the extensibility is EXTENSIBLE or MUTABLE, then the types are assignable. When the reader

receives a sample, the “missing” data (member 'z' in this case) will be set to its default value (0), and the application will be presented with a sample containing 'x', 'y', and 'z'.

6.3 Type Inheritance

The type system now allows the creation of types using inheritance. This applies only to structure types. A structure can be declared to inherit from a parent structure. For example:

```
struct BaseType {
      long  x;
      long  y;
};

struct ChildType {
      long  z;
};
```

For the purposes of type assignability and data processing, an inherited type behaves as if the members of the parent type were expanded directly inside the child type. So, ChildType behaves the same as if it had been declared:

```
struct ChildType {
      long  x;
      long  y;
      long  z;
};
```