RTI Connext DDS
Core Libraries
Extensible Types Guide

Version 6.1.0
Trademarks

RTI, Real-Time Innovations, Connext, NDDS, the RTI logo, 1RTI and the phrase, “Your Systems. Working as one,” are registered trademarks, trademarks or service marks of Real-Time Innovations, Inc. All other trademarks belong to their respective owners.

Copy and Use Restrictions

No part of this publication may be reproduced, stored in a retrieval system, or transmitted in any form (including electronic, mechanical, photocopy, and facsimile) without the prior written permission of Real-Time Innovations, Inc. The software described in this document is furnished under and subject to the RTI software license agreement. The software may be used or copied only under the terms of the license agreement.

This is an independent publication and is neither affiliated with, nor authorized, sponsored, or approved by, Microsoft Corporation.

The security features of this product include software developed by the OpenSSL Project for use in the OpenSSL Toolkit (http://www.openssl.org/). This product includes cryptographic software written by Eric Young (eay@cryptsoft.com). This product includes software written by Tim Hudson (tjh@cryptsoft.com).

Technical Support

Real-Time Innovations, Inc.
232 E. Java Drive
Sunnyvale, CA 94089
Phone: (408) 990-7444
Email: support@rti.com
Website: https://support.rti.com/
### Contents

**Chapter 1 Extensible Types** ................................................................. 1

**Chapter 2 Type Safety and System Evolution**

2.1 Defining Extensible Types ..................................................................... 5
   2.1.1 @id Annotation ........................................................................... 6
   2.1.2 @hashid Annotation .................................................................... 7
   2.1.3 @autoid Annotation .................................................................... 7

2.2 Verifying Type Consistency: Type Assignability .................................. 8

2.3 Type-Consistency Enforcement .............................................................. 11
   2.3.1 Rules For Type-Consistency Enforcement ...................................... 13
   2.3.2 Prevent Type Widening ............................................................... 13
   2.3.3 Type Assignability Properties .................................................... 15

2.4 Verifying Sample Consistency: Sample Assignability ........................... 16

2.5 Notification of Inconsistencies: INCONSISTENT_TOPIC Status ............ 18

2.6 Built-in Topics .................................................................................... 18

**Chapter 3 Type System Enhancements**

3.1 Structure Inheritance ........................................................................... 19

3.2 Optional Members ............................................................................... 20
   3.2.1 Defining Optional Members ....................................................... 20
   3.2.2 Using Optional Members in an Application ................................. 21

3.3 Default Value ...................................................................................... 29
   3.3.1 @default annotation ................................................................ 30
   3.3.2 @default_literal annotation ...................................................... 32

3.4 Ranges ................................................................................................. 33
   3.4.1 Restrictions ............................................................................... 33

**Chapter 4 Data Representation**

4.1 Configuring the CDR ........................................................................... 34
4.1.1 @allowed_data_representation annotation ......................................................... 34
4.2 Extended CDR (encoding version 1) ................................................................. 35
4.3 Extended CDR (encoding version 2) ................................................................. 36
4.4 Choosing the Right Data Representation ......................................................... 37

**Chapter 5 Type Representation**

5.1 TypeObject and TypeCode Type Representation ......................................................... 38
   5.1.1 TypeObject Resource Limits ................................................................. 39
5.2 XML and XSD Type Representations ................................................................. 40

**Chapter 6 TypeCode API Changes** ................................................................. 41

**Chapter 7 DynamicData API** ................................................................. 42

**Chapter 8 ContentFilteredTopics** ................................................................. 43

**Chapter 9 RTI Spy**

9.1 Type Version Discrimination ................................................................. 45

**Chapter 10 Compatibility with Previous Releases** ......................................................... 46
Chapter 1 Extensible Types

This release of Connext DDS includes partial support for the OMG 'Extensible and Dynamic Topic Types for DDS' specification, version 1.3 from the Object Management Group. This support allows systems to define data types in a more flexible way, and to evolve data types over time without giving up portability, interoperability, or the expressiveness of the DDS type system.

Specifically, these are supported:

- Type definitions are now checked as part of the Connext DDS discovery process to ensure that DataReaders will not deserialize the data sent to them incorrectly.
- Type definitions need not match exactly between a DataWriter and its matching DataReaders. For example, a DataWriter may publish a subclass while a DataReader subscribes to a superclass, or a new version of a component may add a field to a preexisting data type.
- Data-type designers can annotate type definitions to indicate the degree of flexibility allowed when the middleware enforces type consistency.
- Type members can be declared as optional, allowing applications to set or omit them in every published sample.
- QoS policies DataRepresentationQosPolicy and TypeConsistencyEnforcementQosPolicy.
- The following builtin-annotations are supported: @id, @hashid, @autoid, @external, @extensibility, @appendable, @mutable, @final, @key, @optional.
- The following builtin-annotations are supported but not propagated as part of the typeobject: @min, @max, @range, @default, @default_literal, @nested, @allowed_data_representation.
- Standard syntax to apply annotations.
- The following fixed-width integer types introduced in Interface Definition Language (IDL) 4.2, shown here with their old type names:
<table>
<thead>
<tr>
<th>IDL 4.2</th>
<th>Old Type Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>int16</td>
<td>short</td>
</tr>
<tr>
<td>int32</td>
<td>long</td>
</tr>
<tr>
<td>int64</td>
<td>long long</td>
</tr>
<tr>
<td>uint16</td>
<td>unsigned short</td>
</tr>
<tr>
<td>uint32</td>
<td>unsigned long</td>
</tr>
<tr>
<td>uint64</td>
<td>unsigned long long</td>
</tr>
</tbody>
</table>

**Note:** You can continue to use the old type names; however, it is preferable to use the new type names because they make the value range explicit.

- Custom annotation definition in IDL. Custom annotations can be defined in IDL, although they are ignored by the middleware (i.e., they will not be part of the typeobject).
- TypeObject v1.
- Extended Common Data Representation (CDR) encoding version 1 and 2.
- The above features are supported in the RTI core middleware in all programming languages except Ada.

The following Extensible Types features are not supported:

- These types: BitMask, BitSet, Map.
- Fixed-width integer types int8 and uint8 are not fully supported, both are always mapped to octets on the wire. However, some language bindings offer support for these types; following is the support by language:
  - In Java, both uint8 and int8 map to a byte, which is signed.
  - In .NET, both uint8 and int8 map to a byte, which is unsigned.
  - In Modern C++, uint8 maps to uint8_t, and int8 maps to int8_t.
  - In C and Traditional C++, uint8 maps to DDS_UINT8, and int8 maps to DDS_INT8.
- Union inheritance
- Custom annotation definition in IDL. Custom annotations can be defined in IDL, although they are ignored by the middleware (i.e., they will not be part of the typeobject).
- TypeObject v2.
- BuiltIn TypeLookup service.
The following builtin-annotations can be defined in IDL, although they will be ignored by the middleware (i.e., they will not be part of the typeobject): verbatim, must_understand, bit_bound, non_serialized, oneway, position, try_construct.

- XML data representation (XML type representation is supported).
- Dynamic language binding compliant with the Extensible Types specification: DynamicType and DynamicData (see DynamicData API (Chapter 7 on page 42)).
- The type member in PublicationBuiltinTopicData and SubscriptionBuiltinTopicData.
- Association of a topic to multiple types within a single DomainParticipant

To see a demonstration of Extensible Types, run RTI Shapes Demo, which can publish and subscribe to two different data types: the "Shape" type or the "Shape Extended" type. If you don't have Shapes Demo installed already, you can download it from https://www.rti.com/free-trial/shapes-demo. If you are not already familiar with how to start Shapes Demo, please see the Shapes Demo User's Manual.

Besides RTI Shapes Demo, several other RTI components include partial support for Extensible Types.
Chapter 2 Type Safety and System Evolution

In some cases, it is desirable for types to evolve without breaking interoperability with deployed components already using those types. For example:

- A new set of applications to be integrated into an existing system may want to introduce additional fields into a structure, or create extended types using inheritance. These new fields can be safely ignored by already deployed applications, but applications that do understand the new fields can benefit from their presence.
- A new set of applications to be integrated into an existing system may want to increase the maximum size of some sequence or string in a Type. Existing applications can receive data samples from these new applications as long as the actual number of elements (or length of the strings) in the received data sample does not exceed what the receiving applications expects. If a received data sample exceeds the limits expected by the receiving application, then the sample can be safely ignored (filtered out) by the receiver.

To support use cases such as the above, the type system introduces the concept of appendable (extensible) and mutable types. A type may be final, appendable (extensible), or mutable:

- **Final**: The type’s range of possible data values is strictly defined. In particular, it is not possible to add elements to members of a collection or aggregated types while maintaining type assignability.
- **Appendable (Extensible)**: Two types, where one contains all of the elements/members of the other plus additional elements/members appended to the end, may remain assignable.
- **Mutable**: Two types may differ from one another with the addition, removal, and/or transposition of elements/members while remaining assignable.

For example, suppose you have:
2.1 Defining Extensible Types

A type’s kind of extensibility is applied with the **Extensibility** annotations seen in Table 2.1 Extensibility Annotations. If you do not specify any particular extensibility, the default is appendable.

**Table 2.1 Extensibility Annotations**

<table>
<thead>
<tr>
<th>IDL</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>@final</td>
<td>struct MyFinalType {</td>
</tr>
<tr>
<td></td>
<td>int32 x;</td>
</tr>
<tr>
<td></td>
<td>};</td>
</tr>
<tr>
<td>@appendable</td>
<td>struct MyExtensibleType {</td>
</tr>
<tr>
<td></td>
<td>int32 x;</td>
</tr>
<tr>
<td></td>
<td>};</td>
</tr>
<tr>
<td>@mutable</td>
<td>struct MyMutableType {</td>
</tr>
<tr>
<td></td>
<td>int32 x;</td>
</tr>
<tr>
<td></td>
<td>};</td>
</tr>
</tbody>
</table>

In this case, if a *DataWriter* writes [1, 2, 3], the *DataReader* will receive [2, 1, 0] (because 0 is the default value of \(x\), which doesn't exist in A's sample).

The type being written and the type(s) being read may differ—maybe because the writing and reading applications have different needs, or maybe because the system and its data design have evolved across versions. Whatever the reason, the databus must detect the differences and mediate them appropriately. This process has several steps:

1. Define what degree of difference is acceptable for a given type.
2. Express your intention for compatibility at run time.
3. Verify that the data can be safely converted.

At run time, the databus will compare the types it finds with the contracts you specified.

```cpp
struct A {
    @id(10) int32 a;
    @id(20) int32 b;
    @id(30) int32 c;
};
```

and

```cpp
struct B {
    @id(20) int32 b;
    @id(10) int32 a;
    @id(40) int32 x;
};
```
## Table 2.1 Extensibility Annotations

<table>
<thead>
<tr>
<th></th>
<th>IDL</th>
<th>XML</th>
<th>XSD</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>@struct</strong></td>
<td>&lt;struct name=&quot;MyFinalType&quot; extensibility=&quot;final&quot;&gt;</td>
<td>&lt;struct name=&quot;MyFinalType&quot;&gt;</td>
<td>&lt;xsd:complexType name=&quot;MyFinalType&quot;&gt;</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>&lt;member name=&quot;x&quot; type=&quot;long&quot;/&gt;</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>&lt;/struct&gt;</td>
<td>&lt;/struct&gt;</td>
<td>&lt;/struct&gt;</td>
</tr>
<tr>
<td><strong>@struct</strong></td>
<td>&lt;struct name=&quot;MyExtensibleType&quot; extensibility=&quot;appendable&quot;&gt;</td>
<td>&lt;struct name=&quot;MyExtensibleType&quot;&gt;</td>
<td>&lt;xsd:complexType name=&quot;MyExtensibleType&quot;&gt;</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>&lt;member name=&quot;x&quot; type=&quot;long&quot;/&gt;</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>&lt;/struct&gt;</td>
<td>&lt;/struct&gt;</td>
<td>&lt;/struct&gt;</td>
</tr>
<tr>
<td><strong>@struct</strong></td>
<td>&lt;struct name=&quot;MyMutableType&quot; extensibility=&quot;mutable&quot;&gt;</td>
<td>&lt;struct name=&quot;MyMutableType&quot;&gt;</td>
<td>&lt;xsd:complexType name=&quot;MyMutableType&quot;&gt;</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>&lt;member name=&quot;x&quot; type=&quot;long&quot;/&gt;</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>&lt;/struct&gt;</td>
<td>&lt;/struct&gt;</td>
<td>&lt;/struct&gt;</td>
</tr>
</tbody>
</table>

Members IDs can be set using the optional @id, @hashid, and @autoid annotations.

### 2.1.1 @id Annotation

The @id annotation allows assigning a 32-bit integer identifier to an element, with the underlying assumption that an identifier should be unique inside its scope of application.

For example:

<table>
<thead>
<tr>
<th></th>
<th>IDL</th>
<th>XML</th>
<th>XSD</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>struct MyType {</strong></td>
<td>&lt;struct name=&quot;MyType&quot;&gt;</td>
<td>&lt;xsd:complexType name=&quot;MyType&quot;&gt;</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>@id(10) int32 x;</td>
<td>&lt;member name=&quot;x&quot; id=&quot;10&quot; type=&quot;int32&quot;/&gt;</td>
<td></td>
</tr>
<tr>
<td></td>
<td>@id(20) int32 y;</td>
<td>&lt;member name=&quot;y&quot; id=&quot;20&quot; type=&quot;int32&quot;/&gt;</td>
<td></td>
</tr>
<tr>
<td><strong>}</strong></td>
<td>&lt;/struct&gt;</td>
<td>&lt;/struct&gt;</td>
<td>&lt;/xsd:complexType&gt;</td>
</tr>
</tbody>
</table>
When not specified, the ID of a member is one plus the ID of the previous one. The first member has ID 0 by default.

```cpp
struct MyType {
    int32 a;
    int32 b;
    @id(100) int32 c;
    int32 d;
};
```

The IDs of 'a', 'b', 'c' and 'd' are 0, 1, 100 and 101.

Member IDs must have a value in the interval [0, 268435455]. The wire representation of mutable or optional members with IDs in the range [0,16128] is more efficient than the wire representation of member IDs in the range [16129, 268435455]. Consequently, the use of IDs in the range [0,16128] is recommended (see Data Representation (Chapter 4 on page 34) for additional details).

### 2.1.2 @hashid Annotation

The `@hashid` annotation provides the value to hash to generate the member ID. If the annotation is used without any parameter or with the empty string as a value then the Member ID will be the hash of the member name.

```IDL
struct HashIdStruct {
    @hashid("hash_text") int32 data;
    int32 data2;
};
```

```xml
<struct name="HashIdStruct">
    <member name="data" hashid="hash_text" type="int32"/>
    <member name="data2" type="int32"/>
</struct>
```

```xsd
<xsd:complexType name="HashIdStruct">
    <xsd:sequence>
        <xsd:element name="data" minOccurs="1" maxOccurs="1" type="xsd:int"/>
        <!-- @hashid hash_text -->
        <xsd:element name="data2" minOccurs="1" maxOccurs="1" type="xsd:int"/>
    </xsd:sequence>
</xsd:complexType>
```

### 2.1.3 @autoid Annotation

The `@autoid` annotation can be applied to modules, structs, or valuetypes and allows you indicate how the identifiers are going to be set for its members.

The values allowed are:

- `@autoid(sequential)`: The next identifier should be computed by incrementing the previous one
- `@autoid(hash)` or `@autoid`: Indicates that the identifiers should be computed with a hashing algorithm based on the name of the member.
2.2 Verifying Type Consistency: Type Assignability

If no annotation is specified, the values will be sequential. The @autoid notation is not supported in XSD when applied to modules.

| IDL       | @autoid
|-----------|--------|
| struct AutoIdStruct {
|    int32 data;
|    int32 data2;
|};

| XML       | <struct name="AutoIdStruct" autoid="hash">
|           |    <member name="data" type="long"/>
|           |    <member name="data2" type="long"/>
|           | </struct>

| XSD       | <xsd:complexType name="AutoIdStruct">
|           |    <xsd:sequence>
|           |        <xsd:element name="data" minOccurs="1" maxOccurs="1" type="xsd:int"/>
|           |        <!- @hashid -->
|           |        <xsd:element name="data2" minOccurs="1" maxOccurs="1" type="xsd:int"/>
|           |        <!- @hashid -->
|           |    </xsd:sequence>
|           | </xsd:complexType>
|           | <![CDATA[@struct true]]>
|           | <!--[CDATA[@autoid hash]]>

2.2 Verifying Type Consistency: Type Assignability

Connext DDS determines if a DataWriter and a DataReader can communicate by comparing the structure of their topic types.

In Connext DDS releases before 5.0.0, the topic types were represented and propagated on the wire using TypeCodes. The Extensible Types specification introduces TypeObjects as the wire representation for a type.

To maintain backward compatibility, Connext DDS can be configured to propagate both TypeCodes and TypeObjects. However, type comparison is only supported with TypeObjects.

Depending on the value for extensibility annotation used when the type is defined, Connext DDS will use a different set of rules to determine if matching shall occur.

If the type extensibility is final, the types will be assignable if they don't add or remove any elements. If they are declared as extensible, one type can have more fields at the end as long as they are not keys.

If the type extensibility is mutable, a type can add, remove or shuffle members in at any position, as long as:

- The type does not add or remove key members
- Members that have the same name also have the same ID, and members that have the same ID also have the same name. (It is possible to change this behavior, see 2.3 Type-Consistency Enforcement on page 11.)

For example, in Table 2.2 Mutable Types Example 1 the middleware can assign MyMutableType1 to or from MyMutableType2, but not to or from MyMutableType3.
2.2 Verifying Type Consistency: Type Assignability

Table 2.2 Mutable Types Example 1

| @mutable struct MyMutableType1 {   | @mutable struct MyMutableType2 {   | @mutable struct MyMutableType3 {   |
|   int32 x;                      |   @id(1) int32 y;                 |   int32 y;                      |
|   int32 y;                      |   @id(2) int32 z;                 |   @key int32 z;                 |
|                                 |   @id(0) int32 x;                 |   int32 x;                      |
|                                 |                                 |                               |
| Note: If you do not explicitly | MyMutableType1 and MyMutableType2 | MyMutableType3 has two issues: |
| declare member IDs, they are   | can be assigned to each other.    | The member IDs x and y do not   |
| assigned automatically starting |                                             | match those of MyMutableType1.  |
| with 0.                         |                                             | For example, the member ID of x |
|                                 |                                             | is 0 in MyMutableType1 but 2   |
|                                 |                                             | in MyMutableType3.              |
|                                 |                                             | MyMutableType3 has an extra key |
|                                 |                                             | member (z).                     |

The type of a member in a mutable type can also change if the new type is assignable. For example, in Table 2.3 Mutable Types Example 2, MyMutableType4 is assignable to or from MyMutableType5 but not to or from MyMutableType6.

Table 2.3 Mutable Types Example 2

| @mutable struct NestedMutableType1 {   | @mutable struct NestedMutableType2 {   | @mutable struct NestedMutableType3 {   |
|   @id(10) int32 a;                  |   @id(20) int16 b;                 |   @id(20) int16 b;                 |
| }                                   |   @id(10) int32 a;                 |   @id(10) int32 a;                 |
|                                       |                                       |                                       |
| struct NestedExtensibleType1 {       | struct NestedExtensibleType2 {      | struct NestedExtensibleType3 {      |
|   string text;                     |   string text;                     |   string text;                     |
| }                                   |                                       |                                       |
| @mutable struct MyMutableType4 {     | @mutable struct MyMutableType5 {     | @mutable struct MyMutableType6 {     |
|   NestedMutableType1 m1;            |   NestedMutableType2 m1;           |   NestedMutableType3 m1;           |
|   NestedExtensibleType1 m2;         |   NestedExtensibleType2 m2;        |   NestedExtensibleType3 m2;        |
| }                                   |                                       |                                       |

MyMutableType6 and MyMutableType4 are not assignable because the types of m1 and m2 are not assignable. NestedExtensibleType3 is just extensible but adds a new member at the beginning. MyMutableType3 changes the type of ‘a’ but the new type (int16) is not assignable.

The member types in an Extensible or Final type can also change as long as the member types are both mutable and assignable. If the new member types are extensible or final, they need to be structurally identical.

If you use CDR encoding version 2 (XCDR2) (see 4.3 Extended CDR (encoding version 2) on page 36), appendable types that are nested into another type can add members at the end of their definition. In the following example, ObservedPosition1 and ObservedPosition2 will not be assignable when using XCDR, but they will be assignable if the encoded version is XCDR2.
Table 2.4 Type Assignability Example

<table>
<thead>
<tr>
<th>@appendable</th>
<th>@appendable</th>
</tr>
</thead>
<tbody>
<tr>
<td>struct Coordinates1 {</td>
<td></td>
</tr>
<tr>
<td>float x;</td>
<td></td>
</tr>
<tr>
<td>float y;</td>
<td></td>
</tr>
</tbody>
</table>
| };
| struct Coordinates2 { |
| float x; |
| float y; |
| float z; // Extra field |
| };
| struct ObservedPosition1 { |
| Coordinates1 position; |
| int64 timestamp; |
| };
| struct ObservedPosition2 { |
| Coordinates2 position; |
| int64 timestamp; |
| };

In the case of union types, it has to be possible, given any possible discriminator value in the DataWriter's type (T2), to identify the appropriate member in the DataReader's type (T1) and to transform the T2 member into the T1 member.

A mutable type that declares a member as optional (see 3.2 Optional Members on page 20) is compatible with a different mutable type that declares the same member as non-optional (the default). This rule does not apply to optional members in final and extensible types.

The following rules apply to other types:

- Primitive types are always final: primitive members cannot change their type.
- Sequences and strings are always mutable: their bounds can change as long as the maximum length in the DataReader type are greater or equal to that of the DataWriter (it is possible to change this behavior, see 2.3 Type-Consistency Enforcement on the next page). A sequence element type can change only if it’s mutable and the new type is assignable.
- Arrays are always final: their bounds cannot change and their element type can only change if it is mutable and the new type assignable.
- Enumerations can be final (they cannot change), extensible (new versions can add constants at the end), or mutable (new versions can add, rearrange or remove constants in any position).

For more information on the rules that determine the assignability of two types, refer to the OMG 'Extensible and Dynamic Topic Types for DDS' specification, version 1.3.

By default, the TypeObjects are compared to determine if they are assignable in order to match a DataReader and a DataWriter of the same topic. You can control this behavior in the DataReader’s TypeConsistencyEnforcementQosPolicy (see 2.3 Type-Consistency Enforcement on the next page).

The DataReader's and DataWriter's TypeObjects need to be available in order to be compared; otherwise their assignability will not be enforced. Depending on the complexity of your types (how many fields, how many different nested types, etc.), you may need to change the default resource limits that control the internal storage and propagation of the TypeObject (see 5.1.1 TypeObject Resource Limits on page 39).
If the logging verbosity is set to NDDS_CONFIG_LOG_VERBOSITY_WARNING or higher, Connext DDS will print a message when a type is discovered that is not assignable, along with the reason why the type is not assignable.

### 2.3 Type-Consistency Enforcement

The `DataReader`'s `TypeConsistencyEnforcementQosPolicy` defines the rules that determine whether the type used to publish a given topic is consistent with that used to subscribe to it.

The QosPolicy structure includes the members in the following table.

#### Table 2.5 DDS_TypeConsistencyEnforcementQosPolicy

<table>
<thead>
<tr>
<th>Type</th>
<th>Field Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DDS_TypeConsistencyKind</td>
<td>kind</td>
<td>Can be any of the following values:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- AUTO_TYPE_COERCION (default)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- ALLOW_TYPE_COERCION</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- DISALLOW_TYPE_COERCION</td>
</tr>
<tr>
<td>DDS_Boolean</td>
<td>ignore_sequence_bounds</td>
<td>Controls whether sequence bounds are taken into consideration for type assignability. If false, a sequence with a larger maximum length may not be assigned to a sequence with a smaller maximum length. If true, sequences and strings in a <code>DataReader</code> type can have a maximum length smaller than that of the <code>DataWriter</code> type. When the length of the sequence in a particular sample is larger than the maximum length, that sample is discarded. Default: true</td>
</tr>
<tr>
<td>DDS_Boolean</td>
<td>ignore_string_bounds</td>
<td>Controls whether string bounds are taken into consideration for type assignability. If false, then a string with a larger maximum length may not be assigned to a string with a smaller maximum length. Default: true</td>
</tr>
<tr>
<td>DDS_Boolean</td>
<td>ignore_member_names</td>
<td>Controls whether member names are taken into consideration for type assignability. If false, members with the same ID and different names are not assignable to each other. If true, members of a type can change their name while keeping their member ID. For example, <code>MyType</code> and <code>MyTypeSpanish</code> are only assignable if <code>ignore_member_names</code> is true: <code>struct MyType { @id(10) int32 x; @id(20) int32 angle; }; struct MyTypeSpanish { @id(10) int32 x; @id(20) int32 angulo; };</code> Default: false</td>
</tr>
</tbody>
</table>
2.3 Type-Consistency Enforcement

### Table 2.5 DDS_TypeConsistencyEnforcementQosPolicy

<table>
<thead>
<tr>
<th>Type</th>
<th>Field Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DDS_Boolean</td>
<td>prevent_type_widening</td>
<td>Controls whether type widening is allowed. A type T2 widens a type T1 when T2 contains required members that are not present in T1. If a DataReader of T2 sets <code>prevent_type_widening</code> to true, then the DataReader will not be matched with a DataWriter of T1 because T1 is not assignable to T2. Default: false</td>
</tr>
<tr>
<td>DDS_Boolean</td>
<td>force_type_validation</td>
<td>Controls whether type information must be available in order to complete matching between a DataWriter and this DataReader. If false, matching may occur as long as the type names match. Note that if the types have the same name but are not assignable, DataReaders may fail to deserialize incoming data samples. Default: false</td>
</tr>
<tr>
<td>DDS_Boolean</td>
<td>ignore_enum_literal_names</td>
<td>Controls whether enumeration constant names are taken into consideration for type assignability. If the option is set to true, then enumeration constants may change their names, but not their values, and still maintain assignability. If the option is set to false, then in order for enumerations to be assignable, any constant that has the same value in both enumerations must also have the same name. For example, enum Color {RED = 0} and enum Color {ROJO = 0} are assignable if and only if <code>ignore_enum_literal_names</code> is true. Default: false</td>
</tr>
</tbody>
</table>

This QoS Policy defines a type consistency **kind**, which allows applications to choose to either allow or disallow data type matching:

- **AUTO_TYPE_COERCION** (default): For a regular DataReader, this default value is translated to ALLOW_TYPE_COERCION. For a Zero Copy DataReader, this default value is translated to DISALLOW_TYPE_COERCION. (See the "Zero Copy Transfer Over Shared Memory" section in the RTI Connext DDS Core Libraries User's Manual for information on why a Zero Copy DataReader requires the DISALLOW_TYPE_COERCION option.)

- **DISALLOW_TYPE_COERCION**: The DataWriter and DataReader must support the same data type in order for them to communicate. (This is the degree of enforcement required by the OMG DDS Specification prior to the OMG ‘Extensible and Dynamic Topic Types for DDS’ Specification.)

- **ALLOW_TYPE_COERCION**: The DataWriter and DataReader need not support the same data type in order for them to communicate as long as the DataReader’s type is assignable from the DataWriter’s type. The concept of assignability is explained in 2.2 Verifying Type Consistency: Type Assignability on page 8.

This policy applies only to DataReaders; it does not have request-offer semantics. The value of the policy cannot be changed after the DataReader has been enabled.

The default enforcement kind is AUTO_TYPE_COERCION. This default kind translates to ALLOW_TYPE_COERCION, except in the following cases:
2.3.1 Rules For Type-Consistency Enforcement

The type-consistency enforcement rules consist of two steps applied on the DataWriter and DataReader side:

- **Step 1.** If both the DataWriter and DataReader specify a TypeObject, it is considered first. If the DataReader allows type coercion, then its type must be assignable from the DataWriter's type, taking into account the values of prevent_type_widening, ignore_sequence_bounds, ignore_string_bounds, ignore_member_names, and ignore_enum_literal_names. If the DataReader does not allow type coercion, then its type must be equivalent to the type of the DataWriter.

- **Step 2.** If either the DataWriter or the DataReader does not provide a TypeObject definition, then the registered type names are examined. The DataReader's and DataWriter's registered type names must match exactly, as was true in Connext DDS releases prior to 5.0. This step will fail if force_type_validation is true, regardless of the type names.

If either Step 1 or Step 2 fails, the Topics associated with the DataReader and DataWriter are considered to be inconsistent (see 2.5 Notification of Inconsistencies: INCONSISTENT_TOPIC Status on page 18).

2.3.2 Prevent Type Widening

The prevent_type_widening field determines whether type widening is allowed. In Figure 2.1: prevent_type_widening = false on the next page, VehicleData_v2 has three members and VehicleData_v1 two members. With type widening allowed, the narrower car (VehicleData_v1, with two members) can write to the wider car (VehicleData_v2), but notice that the DataReader assumes a value that might be misleading (in this case, a default speed of zero).
2.3.2 Prevent Type Widening

Figure 2.1: prevent_type_widening = false

```
@appendable
struct Position {
    double latitude;
    double longitude;
};

@appendable
struct VehicleData_v2 {
    long vehicleID;
    Position position;
    double speed;
}

@appendable
struct VehicleData_v1 {
    long vehicleID;
    Position position;
}
```

DataWriters  Allow Type Widening (Default Behavior)  DataReaders

VehicleData_v1  matches  VehicleData_v2

VehicleData_v1

{1, {37, 122}}

If widening is not allowed (Figure 2.2: prevent_type_widening = true below), VehicleData_v1 and VehicleData_v2 do not communicate with each other.

Figure 2.2: prevent_type_widening = true

```
@appendable
struct Position {
    double latitude;
    double longitude;
};

@appendable
struct VehicleData_v2 {
    long vehicleID;
    Position position;
    double speed;
}

@appendable
struct VehicleData_v1 {
    long vehicleID;
    Position position;
}
```

DataWriters  Disallow Type Widening (Configurable)  DataReaders

VehicleData_v1  does not match  VehicleData_v2

VehicleData_v1

VehicleData_v2

{1, {37, 122}}
2.3.3 Type Assignability Properties

The properties in Table 2.6 Type Assignability Properties relax some of the rules in the standard type-assignability algorithm. These properties can be set in the QoS of the DataReader, DataWriter, and DomainParticipant (in this case all DataReaders and DataWriters created by that DomainParticipant inherit the property). By default they are disabled.

Table 2.6 Type Assignability Properties

<table>
<thead>
<tr>
<th>Property Name</th>
<th>Description</th>
</tr>
</thead>
</table>
| dds.sample_assignability.accept_unknown_union_discriminator | When set to 1, samples containing an unknown union discriminator can be successfully deserialized to the default discriminator value. For example, given the following two types:

**Publisher Type:**
```java
@mutable
union MyUnion switch(int32) {
    case 0:
        int32 m1;
    case 1:
        int16 m2;
    case 2:
        double m3;
};
```

**Subscriber Type:**
```java
@mutable
union MyUnion switch(int32) {
    case 0:
        int32 m1;
    case 1:
        int16 m2;
};
```

By default, if the DataWriter sends a union with the discriminator set to 2, the DataReader cannot deserialize the sample. However, if this property is set to 1, the subscribing application will receive a sample with the discriminator set to 0 and member m1 set to the default value for an int32 (0). The default discriminator value is defined as the default element if one is specified, otherwise the lowest value associated with any discriminator value. The member identified by the default discriminator is also initialized to its default value.

You can set this property as part of the Property QoS for either the DomainParticipant or the DataReader. If it is set in both the DomainParticipant and DataReader, the value in the DataReader’s QoS will be applied.

This functionality is supported both in generated code as well as when using the DynamicData API.
### Table 2.6 Type Assignability Properties

<table>
<thead>
<tr>
<th>Property Name</th>
<th>Description</th>
</tr>
</thead>
</table>
| `dds.sample_assignability.accept_unknown_enum_value` | When set to 1 samples containing an unknown enumerator to be successfully deserialized to the default enumeration value. For example, given the following two types: Publisher Type: ```
enum MyEnum {
  ONE = 1,
  TWO = 2,
  THREE = 3
};
struct MyType {
  MyEnum m1;
};
``` Subscriber Type: ```
enum MyEnum {
  ONE = 1,
  TWO = 2
};
struct MyType {
  MyEnum m1;
};``` By default, if the `DataWriter` sends `m1 = THREE`, the `DataReader` cannot deserialize the sample. However if this property is set to 1 then the Subscribing application will receive a sample with `m1 = ONE`. The default enumeration value is defined as the first declared member of the enumeration. You can set this property as part of the Property QoS for either the `DomainParticipant` or the `DataReader`. If it is set in both the `DomainParticipant` and `DataReader`, the value in the `DataReader`'s QoS will be applied. This functionality is supported both in generated code as well as when using the DynamicData API. |
| `dds.type_consistency.ignore_member_names` | This property has been replaced with `ignore_member_names` and `ignore_enum_literal_names` in the TypeConsistencyEnforcementQosPolicy (see 2.3 Type-Consistency Enforcement on page 11), but is still supported for compatibility with previous releases. If this property is set, its value supersedes the values in the QosPolicy. |
| `dds.type_consistency.ignore_sequence_bounds` | This property has been replaced with `ignore_sequence_bounds` and `ignore_string_bounds` in the TypeConsistencyEnforcementQosPolicy (see 2.3 Type-Consistency Enforcement on page 11), but is still supported for compatibility with previous releases. If this property is set, its value supersedes the values in the QosPolicy. |

### 2.4 Verifying Sample Consistency: Sample Assignability

As described in section 2.2 Verifying Type Consistency: Type Assignability on page 8, Connext DDS determines if a `DataWriter` and a `DataReader` can communicate by comparing the structure of their topic types. When the type published by a `DataWriter` is assignable to the type subscribed by a `DataReader`, the two entities can communicate.

Even if two types are considered assignable, however, some samples may not be assignable. In these cases, the `DataReader` loses the sample. For example, consider a `DataWriter` publishing `Position_v1`, and a `DataReader` subscribing to `Position_v2`:

```
@mutable
struct Position_v1 {
  @range(min=100, max=200) long x;
```
Position_v2 is considered assignable from Position_v1 as both types are structurally the same; however, not all the samples published by the DataWriter will be received by the DataReader. For instance, the DataReader will lose the sample \{x=170,y=100\} and will not provide it to the application because x is outside the valid range [100,150].

When a DataReader loses a sample, Connext DDS logs a warning and updates the SAMPLE_LOST Status with the reason DDS_LOST_BY_DESERIALIZATION_FAILURE.

Another example in which a DataReader may lose samples coming from a DataWriter is when the DataWriter sends a sequence or string with more elements than the DataReader can accept. For example, consider the following types, a DataWriter publishing Poligon_v1 and a DataReader subscribing to Poligon_v2:

```cpp
@mutable
class Position_v2 {
    @range(min=100, max=150) long x;
    @range(min=100, max=150) long y;
};
```

Out of the box, the type Poligon_v1 is assignable to Poligon_v2. Even though the maximum sequence length in Poligon_v2 is smaller than the maximum length in Poligon_v1, ignore_sequence_bounds and ignore_string_bounds are set to TRUE by default on the DataReader TypeConsistencyEnforcementQosPolicy (see 2.3 Type-Consistency Enforcement on page 11).

With ignore_sequence_bounds and ignore_string_bounds set to TRUE, the two types are assignable; however, the DataReader will lose, with the reason DDS_LOST_BY_DESERIALIZATION_FAILURE, samples published with an actual sequence or string length greater than the maximum lengths in Poligon_v2.

If ignore_sequence_bounds and ignore_string_bounds are set to FALSE, the two types will not be assignable.

Note that DataReaders for FlatData types do not deserialize the data and therefore do not drop unassignable samples. (See the “Sending Large Data” chapter in the RTI Connext DDS Core Libraries User's Manual.)
For more information on the rules that determine the assignability of a sample, refer to the column “Object construction” in the assignability tables of the ‘Extensible and Dynamic Topic Types for DDS’ (DDS-Xtypes) specification.

### 2.5 Notification of Inconsistencies: INCONSISTENT_TOPIC Status

Every time a DataReader and DataWriter do not match because the type-consistency enforcement check fails, the INCONSISTENT_TOPIC status is increased.

Notice that the condition under which the middleware triggers an INCONSISTENT_TOPIC status update has changed (starting in release 5.0.0) with respect to previous Connext DDS releases where the change of status occurred when a remote Topic inconsistent with the locally created Topic was discovered.

### 2.6 Built-in Topics

The type consistency value used by a DataReader can be accessed using the type_consistency field in the DDS_SubscriptionBuiltinTopicData (see Table 2.7 New Field in Subscription Builtin Topic Data).

#### Table 2.7 New Field in Subscription Builtin Topic Data

<table>
<thead>
<tr>
<th>Type</th>
<th>New Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DDS_TypeConsistencyEnforcementQosPolicy</td>
<td>type_consistency</td>
<td>Indicates the type_consistency requirements of the remote DataReader (see 2.3 Type-Consistency Enforcement on page 11).</td>
</tr>
</tbody>
</table>

You can retrieve this information by subscribing to the built-in topics and using the DataReader’s get_matched_publication_data() operations.
Chapter 3 Type System Enhancements

3.1 Structure Inheritance

A structure can define a base type as seen in Table 3.1 Base Type Definition in a Structure. Note that when the types are extensible, MyBaseType is assignable from MyDerivedType, and MyDerivedType is assignable from MyBaseType.

Table 3.1 Base Type Definition in a Structure

<table>
<thead>
<tr>
<th>IDL</th>
<th>XML</th>
<th>XSD</th>
</tr>
</thead>
</table>
| struct MyBaseType {
  int32 x;
};
  struct MyDerivedType : MyBaseType {
    int32 y;
};
| <struct name="MyBaseType">
  <member name="x" type="long"/>
</struct>                                      | <xsd:complexType name="MyBaseType">
  <xsd:sequence>
    <xsd:element name="x" minOccurs="1" maxOccurs="1" type="xsd:int"/>
  </xsd:sequence>
</xsd:complexType>                                      | <xsd:complexType name="MyBaseType">
  <xsd:sequence>
    <xsd:element name="y" minOccurs="1" maxOccurs="1" type="xsd:int"/>
  </xsd:sequence>
</xsd:complexType>                                      | <!-- @struct true -->
| struct MyDerivedType : MyBaseType {
    int32 y;
};
| <struct name="MyDerivedType" baseType="MyBaseType">
  <member name="y" type="long"/>
</struct>                                      | <xsd:complexType name="MyDerivedType">
  <xsd:complexContent>
    <xsd:extension base="tns:MyBaseType">
      <xsd:sequence>
        <xsd:element name="y" minOccurs="1" maxOccurs="1" type="xsd:int"/>
      </xsd:sequence>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>                                      | <!-- @struct true -->

Table 3.1 Base Type Definition in a Structure

In Connext DDS 5.0 and higher, value types are equivalent to structures. You can still use the valueType keyword, but using struct is recommended.
3.2 Optional Members

For example:

```c
struct MyType {
    int32 x;
};
valuetype MyType {
    public int32 x;
};
```

The above two types are considered equivalent. Calling the method `equal()` in their TypeCodes will return true. Calling the method `print_IDL()` in the valuette’s TypeCode will print the value type as a struct.

### 3.2 Optional Members

In a structure type, an *optional* member is a member that an application can decide to send or not as part of every published sample.

A subscribing application can determine if the publishing application sent an optional member or not. Note that this is different from getting a default value for a non-optional member that did not exist in the published type (see example in *Type Safety and System Evolution (Chapter 2 on page 4)*), optional members can be explicitly unset.

Using optional members in your types can be useful if you want to reduce bandwidth usage—*Connext DDS* will not send unset optional members on the wire. They are especially useful for designing large sparse types where only a small subset of the data is updated on every write.

This section explains how to define optional members in your types in IDL, XML and XSD and how to use them in applications written in C, C++, Java and in applications that use the DynamicData API. It also describes how optional members affect SQL content filters.

#### 3.2.1 Defining Optional Members

The `@optional` annotation allows you to declare a struct member as optional (see Table 3.2 Declaring Optional Members). If you do not apply this annotation, members are considered non-optional.

In XSD, to declare a member optional, set the `minOccurs` attribute to “0” instead of “1”.

Key members cannot be optional.

<table>
<thead>
<tr>
<th>IDL</th>
<th>XML</th>
</tr>
</thead>
</table>
| struct MyType {
    @optional int32 optional_member;
    int32 non_optional_member;
} | <struct name="MyType">
    <member name="optional_member" optional="true" type="long"/>
    <member name="non_optional_member" type="long"/>
</struct> |
### 3.2.2 Using Optional Members in an Application

This section describes how to use optional members in code generated for C/C++ and Java and with DynamicData API and SQL filters.

#### 3.2.2.1 Using Optional Members in C and the Traditional C++ API

An optional member of type T in a DDS type maps to a pointer-to-T member in a C and C++ struct. Both optional and non-optional strings map to char *.

For example, consider the following IDL type:

```idl
struct Foo {
    string text;
};

struct MyType {
    @optional int32 optional_member1;
    @optional Foo optional_member2;
    int32 non_optional_member;
};
```

This type maps to this C or C++ structure:

```c
typedef struct Foo {
    DDS_Char *text;
} Foo;

typedef struct MyType {
    DDS_Long *optional_member1;
    Foo *optional_member2;
    DDS_Long non_optional_member;
} MyType;
```

An optional member is set when it points to a valid value and is unset when it is NULL. By default, when you create a data sample all optional members are NULL. The TypeSupport API includes the following operations that allow changing that behavior:

```c
MyType *MyTypeTypeSupport_create_data_w_params(
    const struct DDS_TypeAllocationParams_t *alloc_params);
DDS_ReturnCode_t MyTypeTypeSupport_delete_data_w_params(
    struct Foo *a_data,
    const struct DDS_TypeDeallocationParams_t *dealloc_params);
```
Set `alloc_params.allocate_optional_members` to true if you want to have all optional members allocated and initialized to default values.

To allocate or release specific optional string members, use the following functions both in C and traditional C++ without the command-line option `-useStdString`:

- `DDS_String_alloc()`
- `DDS_String_free()`

For traditional C++ code generated using the command line option `-useStdString` use:

- `new ()`
- `delete`

To allocate or release other specific optional members, use the following functions:

In C:

- `DDS_Heap_malloc()`
- `DDS_Heap_calloc()`
- `DDS_Heap_free()`

In traditional C++:

- `new ()`
- `delete`

You can also make an optional member point to an existing variable as long as you set it to `NULL` before deleting the sample.

The following C code shows several examples of how to set and unset optional members when writing samples (note: error checking has been omitted for simplicity):

```c
/* Create and send a sample where all optional members are set */
struct DDS_TypeAllocationParams_t allocParams = DDS_TYPE_ALLOCATION_PARAMS_DEFAULT;
allocParams.allocate_optional_members = DDS_BOOLEAN_TRUE;
MyType *sample = MyTypeTypeSupport_create_data_w_params(&allocParams);
*sample->optional_member1 = 1;
strcpy(sample->optional_member2->text, "hello");
```
3.2.2 Using Optional Members in an Application

```c
sample->non_optional_member = 2;
MyTypeDataWriter_write(
    MyType_writer,
    instance,
    DDS_HANDLE_NIL);

/* This time, don't send optional_member1 */
DDS_Heap_free(sample->optional_member1);
sample->optional_member1 = NULL;
MyTypeDataWriter_write(MyType_writer, sample, DDS_HANDLE_NIL);

/* Delete the sample */
retcode = MyTypeTypeSupport_delete_data_ex(sample, DDS_BOOLEAN_TRUE);

/* Create and send a sample where all optional members are unset */
sample = MyTypeTypeSupport_create_data_ex(DDS_BOOLEAN_FALSE);
sample->non_optional_member = 3;
MyTypeDataWriter_write(MyType_writer, sample, DDS_HANDLE_NIL);

/* Now send optional_member1 */
sample->optional_member1 = (DDS_Long *)DDS_Heap_malloc(sizeof(DDS_Long));
*sample->optional_member1 = 1;
sample->non_optional_member = 3;
MyTypeDataWriter_write(MyType_writer, sample, DDS_HANDLE_NIL);

/* Delete the sample */
retcode = MyTypeTypeSupport_delete_data_ex(sample, DDS_BOOLEAN_TRUE);
```

And this example shows how to read samples that contain optional members in C:

```c
/* Create a sample (no need to allocate optional members here) */
struct DDS_SampleInfo info;
MyType *sample = MyTypeTypeSupport_create_data();

/* Read or take as usual */
MyTypeDataReader_take_next_sample(MyType_reader, sample, &info);
if (info.valid_data)
{
    printf("optional_member 1");
    if (sample->optional_member1 != NULL)
    {
        printf(" = %d", *sample->optional_member1);
    }
    else
    {
        printf("is not set 
");
    }
    printf("non_optional_member = %d", sample->non_optional_member);
}
MyTypeTypeSupport_delete_data(sample);
```

The following C++ code shows several examples of how to set and unset optional members when writing samples (note: error checking has been omitted for simplicity):
3.2.2 Using Optional Members in an Application

// Create and send a sample where all optional members are set
MyType *sample = MyTypeTypeSupport::create_data(
    DDS_TypeAllocationParams_t().set_allocate_optional_members(
        DDS_BOOLEAN_TRUE));
*sample->optional_member1 = 1;
strcpy(sample->optional_member2->text, "hello");
sample->non_optional_member = 2;
writer->write(*sample, DDS_HANDLE_NIL);

// This time, don't send optional_member1
delete sample->optional_member1;
sample->optional_member1 = NULL;
writer->write(*sample, DDS_HANDLE_NIL);

// Delete the sample
MyTypeTypeSupport::delete_data(sample);

// Create and send a sample where all optional members are unset
sample = MyTypeTypeSupport::create_data();
sample->non_optional_member = 3;
writer->write(*sample, DDS_HANDLE_NIL);

// Now send optional_member1:
sample->optional_member1 = new DDS_Long();
*sample->optional_member1 = 4;
writer->write(*sample, DDS_HANDLE_NIL);

// Delete the sample
MyTypeTypeSupport::delete_data(sample);

And this example shows how to read samples that contain optional members in traditional C++:

// Create a sample (no need to allocate optional members here)
DDS_SampleInfo info;
sample = MyTypeTypeSupport::create_data();

// Read or take as usual
reader->take_next_sample(*sample, info);
if (info.valid_data)
{
    std::cout << "optional_member1 ";
    if (sample->optional_member1 != NULL)
    {
        std::cout << "= " << *sample->optional_member1 << "\n";
    }
    else
    {
        std::cout << "is not set\n";
    }
    std::cout << "non_optional_member = " << sample->non_optional_member << "\n";
}

// Delete the sample
MyTypeTypeSupport::delete_data(sample);
3.2.2 Using Optional Members in an Application

3.2.2.2 Using Optional Members in the Modern C++ API

An optional member of type \( T \) in a DDS type maps to the value-type \( \text{dds::core::optional}<T> \) in the modern C++ API.

For example, consider the following IDL type:

```idl
struct MyType {
    @optional int32 optional_member1;
    @optional Foo optional_member2;
    int32 non_optional_member;
};
```

This type maps to this C++ class:

```cpp
class NDDSUSERDllexport MyType {
public:
    // ...
    dds::core::optional<int32_t>& optional_member1();
    const dds::core::optional<int32_t>& optional_member1() const;
    void optional_member1(const dds::core::optional<int32_t>& value);
    dds::core::optional<Foo>& optional_member2();
    const dds::core::optional<Foo>& optional_member2() const;
    void optional_member2(const dds::core::optional<Foo>& value);
    int32_t non_optional_member() const;
    void non_optional_member(int32_t value);
    // ...
};
```

By default optional members are unset (\( \text{dds::core::optional}<T>::\text{is\_set}() \) is false). To set an optional member, simply assign a value; to reset it use \text{reset()} or assign a default-constructed \text{optional}<T>:

```cpp
MyType sample; // all optional members created unset
sample.optional_member1() = 5; // now sample.optional_member1().is_set() == true
sample.optional_member1(5); // alternative way of setting the optional member
sample.optional_member2() = Foo/* ... */;
sample.optional_member1().reset(); // now sample.optional_member1().is_set == false
sample.optional_member1() = dds::core::optional<int32_t>(); // alternative way of resetting the optional member
```

To get the value by reference, use \text{get()}:

```cpp
int x = sample.optional_member1().get(); // if !is_set(), throws
.dds::core::PreconditionNotMetError.
sample.optional_member2().get().foo_member(10);
```

Note that \( \text{dds::core::optional} \) manages the creation, assignment and destruction of the contained value, so unlike the traditional C++ API you don't need to reserve and release a pointer.

3.2.2.3 Using Optional Members in Java

Optional members have the same mapping to Java class members as non-optional members, except that \text{null} is a valid value for an optional member. Primitive types map to their corresponding Java wrapper classes (to allow nullifying).
Generated Java classes also include a `clear()` method that resets all optional members to null.

For example, consider the following IDL type:

```idl
struct MyType {
    @optional int32 optional_member1;
    @optional Foo optional_member2;
    int32 non_optional_member;
};
```

This type maps to this Java class:

```java
class MyType {
    public Integer optional_member1 = null;
    public Foo optional_member2 = null;
    public int non_optional_member = 0;
    // ...
    public void clear() { /* ... */ }
    // ...
}
```

An optional member is set when it points to an object and is unset when it is `null`.

The following code shows several examples on how to set and unset optional members when writing samples:

```java
// Create and send a sample with all the optional members set
MyType data = new MyType(); // All optional members are null
data.optional_member1 = 1; // Implicitly converted to Integer
data.optional_member2 = new Foo(); // Create Foo object
data.optional_member2.text = "hello";
data.non_optional_member = 2;
writer.write(data, InstanceHandle_t.HANDLE_NIL);

// This time, don't send optional_member1
data.optional_member1 = null;
writer.write(data, InstanceHandle_t.HANDLE_NIL);

// Send a sample where all the optional members are unset
data.clear(); // Set all optional members to null
data.non_optional_member = 3;
writer.write(data, InstanceHandle_t.HANDLE_NIL);

// Now send optional_optional_member
data.optional_member1 = 4;
writer.write(data, InstanceHandle_t.HANDLE_NIL);
```

And this example shows how to read samples that contain optional members:

```java
// Create a sample
MyType data = new MyType();
SampleInfo info = new SampleInfo();

// Read or take as usual
reader.take_next_sample(data, info);
if (info.valid_data) {
```
3.2.2 Using Optional Members in an Application

```java
System.out.print("optional_member1 ");
if (data.optional_member1 != null) {
    System.out.println(" = " + data.optional_member1);
} else {
    System.out.println(" is unset");
}
System.out.println("non_optional_member = " + data.non_optional_member);
```

3.2.2.4 Using Optional Member with DynamicData

This version of Connext DDS supports a pre-standard version of DynamicData (see DynamicData API (Chapter 7 on page 42)). However it does support optional members.

Any optional member can be set with the regular setter methods in the DynamicData API, such as `DDS_DynamicData::set_long()`. An optional member is considered unset until a value is explicitly assigned using a ‘set’ operation.

To unset a member, use `DDS_DynamicData::clear_optional_member()`.

The C and C++ ‘get’ operations, such as `DDS_DynamicData::get_long()`, return DDS_RETCODE_NO_DATA when an optional member is unset; in Java, the ‘get’ methods throw a RETCODE_NO_DATA exception.

The following C++ example shows how to set and unset optional members before writing a sample. The example uses the same type (MyType) as in previous sections. This example assumes you already know how to use the DynamicData API, in particular how to create a DynamicDataTypeSupport and a DynamicData topic. More information and examples are available in the API Reference HTML documentation (select Modules, RTI Connext DDS API Reference, Topic Module, Dynamic Data).

```cpp
// Note: error checking omitted for simplicity
DDS_DynamicData * data = type_support.create_data();

// Set all optional members and write a sample
data->set_long("optional_member1",
    DDS_DYNAMIC_DATA_MEMBER_ID_UNSPECIFIED, 1);

// Bind optional_member2 and set the text field
data->bind_complex_member(optionalMember2, "optional_member2",
    DDS_DYNAMIC_DATA_MEMBER_ID_UNSPECIFIED); optionalMember2.set_string("text",
    DDS_DYNAMIC_DATA_MEMBER_ID_UNSPECIFIED, "hello");
data->unbind_complex_member(optionalMember2);
data->set_long("non_optional_member",
    DDS_DYNAMIC_DATA_MEMBER_ID_UNSPECIFIED, 2);
writer->write(*data, DDS_HANDLE_NIL);

// This time, don't send optional_member1
data->clear_optional_member("optional_member1",
    DDS_DYNAMIC_DATA_MEMBER_ID_UNSPECIFIED);
writer->write(*data, DDS_HANDLE_NIL);
```
In this example we read samples that contain optional members:

```cpp
DDS_SampleInfo info;
DDS_DynamicData * data = type_support->create_data();
reader->take_next_sample(*data, info);
if (info.valid_data) {
    DDS_Long value;
    DDS_ReturnCode_t retcode = data->get_long(value, 
        "optional_member1",
        DDS_DYNAMIC_DATA_MEMBER_ID_UNSPECIFIED);
    if (retcode == DDS RETCODE_OK)
    {
        std::cout << "optional_member1 = " << value << 
    } else if (retcode == DDS RETCODE_NO_DATA)
    {
        std::cout << "optional_member1 is not set\n";
    } else 
    {
        std::cout << "Error getting optional_member1\n";
    }
    retcode = data->get_long(value, "non_optional_member",
        DDS_DYNAMIC_DATA_MEMBER_ID_UNSPECIFIED);
    if (retcode == DDS RETCODE_OK) 
    {
        std::cout << "non_optional_member = " << value << 
    } else 
    {
        std::cout << "Error getting non_optional_member\n";
    }
} // Delete the sample
type_support->delete_data(data);
```

### 3.2.2.5 Using Optional Members in SQL Filter Expressions

SQL filter expressions used in ContentFilteredTopics and QueryConditions (see `ContentFilteredTopics (Chapter 8 on page 43)` in this document and Section 4.6.7 (ReadConditions and QueryConditions) and Section 5.4 (ContentFilteredTopics) in the *RTI Connext DDS Core Libraries User's Manual*) can refer to optional members. The syntax is the same as for any other member.

For example, given the type `MyType`:

```cpp
struct Foo {
    string text;
};
struct MyType {
    @optional int32 optional_member1;
    @optional Foo optional_member2;
    int32 non_optional_member;
};
```

This is a valid SQL filter expression:

`"optional_member1 = 1 AND optional_member2.text = ‘hello’ AND non_optional_member = 2"`

Any comparison involving an optional member (\(=\), \(<\), \(\le\), \(\ge\), or \(\ne\)) evaluates to false if the member is unset.
For example, both “optional_member1 <> 1” and “optional_member1 = 1” will evaluate to false if optional_member1 is unset; however “optional_member1 = 1 OR non_optional_member = 1” will be true if non_optional_member is equal to 1 (even if optional_member1 is unset). The expression “optional_member2.text = ‘hello’” will also be false if optional_member2 is unset.

3.3 Default Value

If the value for an optional member is not provided on the wire, the member is initialized to NULL. For non-optional members, the member is considered to have the default value defined in Table 3.3 Default Values for Non-Optional Members from XTypes Specification.

Table 3.3 Default Values for Non-Optional Members from XTypes Specification, taken from the "Extensible and Dynamic Topic Types for DDS" (DDS-XTypes) specification, describes the default values for non-optional members.

Table 3.3 Default Values for Non-Optional Members from XTypes Specification

<table>
<thead>
<tr>
<th>Type Kind</th>
<th>Default Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>BYTE</td>
<td>0x00</td>
</tr>
<tr>
<td>BOOLEAN</td>
<td>FALSE</td>
</tr>
<tr>
<td>INT_16_TYPE, UINT_16_TYPE, INT_32_TYPE, UINT_32_TYPE, INT_64_TYPE, UINT_64_TYPE, FLOAT_32_TYPE, FLOAT_64_TYPE, FLOAT_128_TYPE</td>
<td>0</td>
</tr>
<tr>
<td>CHAR_8_TYPE, CHAR_16_TYPE</td>
<td>'0'</td>
</tr>
<tr>
<td>STRING_TYPE</td>
<td>&quot;&quot;</td>
</tr>
<tr>
<td>ARRAY_TYPE</td>
<td>An array of the same dimensions and same element type whose elements take the default value for their corresponding type.</td>
</tr>
<tr>
<td>ALIAS_TYPE</td>
<td>The default type of the alias's base type.</td>
</tr>
<tr>
<td>SEQUENCE_TYPE</td>
<td>A zero-length sequence of the same element type.</td>
</tr>
<tr>
<td>MAP_TYPE</td>
<td>An empty map of the same element type.</td>
</tr>
<tr>
<td>ENUM_TYPE</td>
<td>The first value in the enumeration.</td>
</tr>
<tr>
<td>UNION_TYPE</td>
<td>A union with the discriminator set to select the default element, if one is defined, or otherwise to the lowest value associated with any member. The value of that member set to the default value for its corresponding type.</td>
</tr>
<tr>
<td>STRUCTURE_TYPE</td>
<td>A structure without any of the optional members and with other members set to their default values based on their corresponding types.</td>
</tr>
</tbody>
</table>
3.3.1 @default annotation

This annotation allows you to specify a default value for a primitive, enum, or string member. It overwrites the default value in Table 3.3 Default Values for Non-Optional Members from XTypes Specification. For example:

```c
struct Position {
    int32 x;
    @default(70) int32 y;
    @default(80) int32 z;
};
```

In the above example, when a new Position data object is created (TypeSupport::create_data, for example), the members y and z will get the values 50 and 70 respectively, while the member x will get the default 0.

The members will also get the same default values when they are not received on the wire. For example, assume a DataWriter publishing PubPosition and a DataReader subscribing to Position:

```c
struct PubPosition {
    int32 x;
};
```

Position is assignable from PubPosition according to the assignability rules described in Chapter 2 Type Safety and System Evolution on page 4. When the DataReader receives a new sample from the DataWriter, the members y and z (not present on the wire) will get the values 70 and 80.

The default annotation can be applied to members with the following types: boolean, octet, int16, uint16, int32, uint32, int64, uint64, float, double, char, wchar, string, wstring, and enums. The annotation is not currently supported for long double members.

The default annotation can also be applied to aliases of the previous types. For example:

```c
typedef int32 XCoordinate;  
@default(70)
typedef int32 YCoordinate;  
@default(80)
typedef int32 ZCoordinate;

struct Position {
    XCoordinate x;
    YCoordinate y;
    ZCoordinate int32 z;
};
```

The advantage of assigning a default to Alias types is that you do not have to duplicate the annotation value in every structure using coordinates.

The value in the @default annotation can refer to constants declared in the IDL file and can contain expressions using the constants. For example:
3.3.1 @default annotation

```c
const int32 Y_DEFAULT = 70;
const int32 Z_DEFAULT = 79;
struct Position {
    int32 x;
    @default(Y_DEFAULT) int32 y;
    @default(Z_DEFAULT + 1) int32 z;
};
```

### 3.3.1.1 Restrictions

- The default annotation cannot be applied to optional and external members even if their types are types in which the annotation is supported. For example:

```c
struct Position {
    int32 x;
    @default(70) int32 y;
    @default(80) @optional int32 z; // Not supported. Code generation error
};
```

- The default annotation is not currently supported on arrays and sequences even if their types are types in which the annotation is supported. For example:

```c
struct Positions {
    int32 x[1024];
    @default(50)
    int32 y[1024]; // Not supported. Code generation error
    @default(80)
    int32 z[1024]; // Not supported. Code generation error
};
```

A workaround for this limitation is to encapsulate the primitive members into a structure. For example:

```c
struct Position {
    int32 x;
    @default(70) int32 y;
    @default(80) int32 z;
};
struct Positions {
    Position position[1024];
};
```

- The default annotation value for a uint64 type cannot refer to a constant. For example:

```c
const uint64 MY_UINT64 = 9223372036854775808;
struct Example {
    @default(MY_UINT64)
    uint64 x;
};
```

A workaround for this limitation is to not use the constant but the literal value instead. For example:
Expressions are not supported when converting to XSD for the following types:

- Long long
- Unsigned long long
- Float
- Double
- Long double

The TypeCode API (DynamicType API in Modern C++) does not provide a public API to obtain the value of the default annotation.

### 3.3.2 @default_literal annotation

By default, the default value of an enumeration corresponds to the first value of the enumeration. In the following example, the default value is GREEN:

```cpp
enum Color {
    GREEN,
    RED,
    BLUE
};
```

The annotation `@default_literal` can be used to select a different enumerator as the default value. In the following example, the default value is RED:

```cpp
enum Color {
    GREEN,
    @default_literal RED,
    BLUE
};
```

The default value for an enumeration can be overwritten for a structure/union member referring to this enumeration using the `@default` annotation. For example:

```cpp
enum Color {
    GREEN,
    @default Literal RED,
    BLUE
};

struct Shape {
    @default(BLUE)
    Color shape_color;
}
3.3.2.1 Restrictions

The TypeCode API (DynamicType API in Modern C++) does not provide a public API to obtain the value of the default_literal annotation.

3.4 Ranges

The annotations @range, @min, and @max can be used to restrict the possible values for a primitive member. For example:

```cpp
struct Position {
    @range(min = 0, max = 200) int32 x;
    @min(50) @default(70) int32 y;
    @max(200) @default(80) int32 z;
};
```

The annotations are enforced at serialization/deserialization time, not when the value of an object is set. For example, assume the following Position: \{x= -3, y= 60, z = 150\}. If you try to publish a Position sample with this value, the `DataWriter::write` operation will fail with DDS_RETCODE_ERROR. If a `DataReader` receives this sample, the sample will be lost with the reason DDS_LOST_BY_DESERIALIZATION_FAILURE and it will not be provided to the application. In both cases, you will see a log message indicating that x was outside its valid range.

The range annotations can be applied to the following types: octet, int16, uint16, int32, uint32, int64, uint64, float, double. These annotations are not supported in long double.

If you specify a @default value for a member that is outside the valid range, the code generation will fail. For example:

```cpp
struct Position {
    @default(300) @range(min = 0, max = 200) int32 x; // Failure. Default outside valid range
    @min(50) @default(70) int32 y;
    @max(200) @default(80) int32 z;
};
```

3.4.1 Restrictions

- For performance reasons, the range annotations are not currently applied to samples of types marked with @language_binding(FLAT_DATA). The annotations can be used for the type members, but they are only informational.
- The TypeCode API (DynamicType API in Modern C++) does not provide a public API to obtain the value of the @range, @min, and @max annotations.
Chapter 4 Data Representation

The data representation specifies the ways in which a data sample of a given type are communicated over the network.

The OMG 'Extensible and Dynamic Topic Types for DDS' specification, version 1.3 defines three different data representations:

- Extended Common Data Representation (CDR) encoding version 1 (XCDR)
- Extended CDR encoding version 2 (XCDR2)
- XML data representation

Connext DDS 6.0.0 and above implements both XCDR and XCDR2. Connext DDS 5.3.1 and below implements only XCDR. XML data representation is not supported.

4.1 Configuring the CDR

You may use the DataRepresentationQosPolicy in the DataWriterQos to configure which version of Extended CDR, version 1 or version 2, the DataWriter will use to serialize its data. The same QosPolicy exists in the DataReaderQos to configure which version(s) the DataReader will accept from DataWriters. DataWriters can offer only one data representation, while DataReaders can request multiple data representations.

For more information, see "DATA_REPRESENTATION QosPolicy" in the RTI Connext DDS Core Libraries User's Manual.

4.1.1 @allowed_data_representation annotation

The data representations that you are allowed to configure in the DataRepresentationQosPolicy for a type ‘T’ are limited to the allowed data representations for the type.

The @allowed_data_representation annotation lets you restrict the data representations that may be used to encode a data object of a specific type. (You can select from this restricted set when setting
the DataRepresentationQosPolicy.) The IDL definition of the @allowed_data_representation annotation is as follows:

```idl
// Positions are defined to match the values of the DataRepresentationId_t
// XCDR_DATA_REPRESENTATION, XML_DATA_REPRESENTATION, and
// XCDR2_DATA_REPRESENTATION
@bit_bound(32)
bitmask DataRepresentationMask {
    @position(0) XCDR,
    @position(1) XML,
    @position(2) XCDR2
}

@annotation allowed_data_representation {
    DataRepresentationMask value;
};
```

For example:

```idl
@allowed_data_representation(XCDR2)
struct Position {
    int32 x;
    int32 y;
};
```

DataWriters and DataReaders using the previous type can publish and subscribe to only an XCDR2 representation, regardless of the value set in the DataRepresentationQosPolicy. (If a DataWriter or DataReader in this case sets its DataRepresentationQosPolicy to XCDR, Connext DDS will automatically change it to XCDR2 and print a log message indicating this change.)

If the @allowed_data_representation annotation is not present, Connext DDS interprets the data representation as if the DataRepresentationMask value was set to XCDR|XCDR2 for PLAIN language binding and XCDR2 for FLAT DATA language binding. For information about the RTI FlatData™ language binding, see the "Sending Large Data" chapter in the RTI Connext DDS Core Libraries User's Manual.

### 4.2 Extended CDR (encoding version 1)

The "traditional" OMG CDR (PLAIN_CDR) is used for final and extensible types. It is also used for primitive, string, and sequence types.

Mutable types and optional members use parameterized CDR (PL_CDR), in which each member is preceded by a member header that consists of the member ID and member serialized length.

The member header can be 4 bytes (2 bytes for the member ID and 2 bytes for the serialized length) or 12 bytes (where 4 bytes are used for the member ID and 4 bytes are used for the length).

Member IDs greater than 16,128 require a 12-byte header. Therefore, to reduce network bandwidth, the recommendation is to use member IDs less than or equal to 16,128.
Also, members with a serialized size greater than 65,535 bytes require a 12-byte header. Notice that for members with a member ID less than 16,129 and a serialized size less than 65,536 bytes, it is up to the implementation to decide whether or not to use a 12-byte header. For this version of Connext DDS, the header selection rules are as follows:

- If the member ID is greater than 16,128, use a 12-byte header.
- Otherwise, if the member is a primitive type (int16, uint16, int32, uint32, int64, uint64, float, double, long double, boolean, octet, char), use a 4-byte header.
- Otherwise, if the member is an enumeration, use a 4-byte header.
- Otherwise, if the maximum serialized size of the type is less than 65,536 bytes, use a 4-byte header.
- Otherwise, use a 12-byte header.

### 4.3 Extended CDR (encoding version 2)

From the ‘Extensible and Dynamic Topic Types for DDS’ specification:

*The specification defines three encoding formats used with encoding version 2: PLAIN_CDR2, DELIMITED_CDR, and PL_CDR2.*

- **PLAIN_CDR2** shall be used for all primitive, string, and enumerated types. It is also used for any type with an extensibility kind of FINAL. The encoding is similar to PLAIN_CDR except that INT64, UINT64, FLOAT64, and FLOAT128 are serialized into the CDR buffer at offsets that are aligned to 4 [bytes] instead of 8 ....
- **DELIMITED_CDR** shall be used for types with an extensibility kind of APPENDABLE. It serializes a UINT32 delimiter header (DHEADER) before serializing the object using PLAIN_CDR2. The delimiter encodes the endianness and the length of the serialized object that follows.
- **PL_CDR2** shall be used for *aggregated types* with an extensibility kind of MUTABLE. Similar to DELIMITED_CDR, it also serializes a DHEADER before serializing the object. In addition, it serializes a member header (EMHEADER) ahead of each serialized member. The member header encodes the member ID, the must-understand flag, and the length of the serialized member that follows.

In Extended CDR encoding version 2, wchar sizes changed from 4 bytes (Char32) to 2 bytes (Char16). For more information about encoding version 2, please see the [OMG ‘Extensible and Dynamic Topic Types for DDS’ specification, version 1.3](https://www.omg.org/spec/DDS-ECM/1.3/).
4.4 Choosing the Right Data Representation

Extended CDR encoding 2 (XCDR2) is more efficient on the wire than Extended CDR encoding 1 (XCDR). For new applications, Extended CDR encoding 2 is the recommended data representation; however, if you need to keep compatibility and interoperability with old Connext DDS applications (5.3.1 and below), you may have to continue using Extended CDR encoding 1.

*DataReaders* can be configured to receive data using both XCDR2 and XCDR. This way, a *DataReader* can still interoperate and receive data from old *Connext DDS DataWriters* using XCDR, while receiving data from new *DataWriters* using XCDR2.

The opposite is not true. *DataWriters* can publish only one data representation. Therefore, if there is a requirement to receive data for a topic 'T' with old *Connext DDS DataReaders*, you will have to continue to publish data for topic 'T' with XCDR representation on the new *DataWriters* or use a bridge such as *Routing Service* to translate between XCDR and XCDR2.
Chapter 5 Type Representation

The type representation specifies the ways in which a type can be externalized so that it can be stored in a file or sent over the network.

The OMG 'Extensible and Dynamic Topic Types for DDS' specification, version 1.3 describes four different type representations: IDL, TypeObject, XML, and XSD.

5.1 TypeObject and TypeCode Type Representation

Earlier versions of Connext DDS (4.5f and lower) used TypeCodes as the wire representation to communicate types over the network and the TypeCode API to introspect and manipulate the types at run time.

The Extensible Types specification uses TypeObjects as the wire representation and the DynamicType API to introspect and manipulate the types. Types are propagated by serializing the associated TypeObject representation.

This release does not enable TypeCode propagation by default, but to maintain backward compatibility with previous releases it can be enabled; see the section, "TypeCode information is not sent by default" in the Migration Guide on the RTI Community Portal (https://community.rti.com/documentation). Support for the TypeCodes feature may be discontinued in future releases.

Connext DDS 5.x and higher supports TypeObjects v1 as the wire representation. (TypeObjects v2, which were introduced in Extensible and Dynamic Topic Types for DDS 1.2, are not supported.)

In this release, only Modern C++ supports the DynamicType API to introspect the types at runtime. Other language bindings must use the TypeCode API.

You can introspect the discovered type independently of the wire format by using the type_code member in the PublicationBuiltinTopicData and SubscriptionBuiltinTopicData structures for all language bindings but Modern C++. In Modern C++, the type information can be accessed using the type() or get_type_no_copy() accessors.
One important limitation of using TypeCodes as the wire representation is that their serialized size is limited to 65 KB. This is a problem for services and tools that depend on the discovered types, such as RTI Routing Service and RTI Spreadsheet Add-in for Microsoft Excel. With the introduction of TypeObjects, this limitation is removed since the size of the serialized representation is not bounded.

To summarize:

<table>
<thead>
<tr>
<th>Wire Representation</th>
<th>Connext DDS 5.x and Higher</th>
<th>Connext DDS 4.5f and Earlier</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>TypeObjects or TypeCodes (for backwards compatibility)</td>
<td>TypeCodes</td>
</tr>
<tr>
<td>For Introspection at Run Time</td>
<td>TypeCode API (DynamicType API for Modern C++)</td>
<td>TypeCode API</td>
</tr>
<tr>
<td>Maximum Size of Serialized Representation</td>
<td>When using TypeObjects: Unbounded When using TypeCodes: 65 KB</td>
<td>65 KB</td>
</tr>
</tbody>
</table>

### 5.1.1 TypeObject Resource Limits

Table 5.1 TypeObject Fields in DomainParticipantResourceLimitsQosPolicy lists fields in the DomainParticipantResourceLimitsQosPolicy that control resource utilization when the TypeObjects in a DomainParticipant are stored and propagated.

Note that memory usage is optimized; only one instance of a TypeObject will be stored, even if multiple local or remote DataReaders or DataWriters use it.

#### Table 5.1 TypeObject Fields in DomainParticipantResourceLimitsQosPolicy

<table>
<thead>
<tr>
<th>Type</th>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DDS_Long</td>
<td>type_object_max_serialized_length</td>
<td>The maximum length, in bytes, that the buffer to serialize a TypeObject can consume.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>This parameter limits the size of the TypeObject that a DomainParticipant is able to propagate. Since TypeObjects contain all of the information of a data structure, including the strings that define the names of the members of a structure, complex data structures can result in TypeObjects larger than the default maximum of 3072 bytes. This field allows you to specify a larger value. Cannot be UNLIMITED. Default: 8192</td>
</tr>
<tr>
<td>DDS_Long</td>
<td>type_object_max_deserialized_length</td>
<td>The maximum number of bytes that a deserialized TypeObject can consume. This parameter limits the size of the TypeObject that a DomainParticipant is able to store. Default: UNLIMITED</td>
</tr>
<tr>
<td>DDS_Long</td>
<td>deserialized_type_object_dynamic_allocation_threshold</td>
<td>A threshold, in bytes, for dynamic memory allocation for the deserialized TypeObject. Above it, the memory for a TypeObject is allocated dynamically. Below it, the memory is obtained from a pool of fixed-size buffers. The size of the buffers is equal to this threshold. Default: 4096</td>
</tr>
</tbody>
</table>
The TypeObject is needed for type-assignability enforcement.

Since TypeObjects contain all of the information of a data structure, including the strings that define the names of the members of a structure, complex data structures can result in large TypeObjects that frequently require enabling asynchronous publication for discovery data.

To reduce bandwidth usage during discovery for large TypeObjects, Connext DDS allows compressing the TypeObject information. Compression is enabled by default, and it can be configured using the QoS value DDS_DiscoveryConfigQosPolicy::endpoint_type_object_lb_serialization_threshold. For additional information, see the section “DISCOVERY_CONFIG QosPolicy” in the RTI Connext DDS Core Libraries User's Manual.

By default, Connext DDS 5.3.1 and lower propagated both the pre-standard TypeCode and the TypeObject. Connext DDS 6.0.0 and higher only propagates TypeObjects by default. You can change this behavior:

<table>
<thead>
<tr>
<th>To propagate TypeObject only:</th>
<th>Set type_code_max_serialized_length = 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>To propagate TypeCode only:</td>
<td>Set type_object_max_serialized_length = 0</td>
</tr>
<tr>
<td>To propagate none:</td>
<td>Set type_code_max_serialized_length = 0 and type_object_max_serialized_length = 0</td>
</tr>
<tr>
<td>To propagate both (default):</td>
<td>Use the default values of type_code_max_serialized_length and type_object_max_serialized_length or modify them if the type size requires so.</td>
</tr>
</tbody>
</table>

### 5.2 XML and XSD Type Representations

The XML and XSD type-representation formats available in Connext DDS formed the basis for the DDS-XTypes specification of these features.

The XML format is compatible with the format described in the XTypes specification.

The XSD format, however, has not been completely updated to the new standard format. For example, in Connext DDS, built-in annotations are applied using comments, whereas in the XTypes specification they are applied using <xsd:annotation>.

For additional information on how to apply built-in annotations using XSD Type Representation in Connext DDS see the section "Creating User Data Types with XML Schemas (XSD)" in the RTI Connext DDS Core Libraries User's Manual.

For additional information on how to apply built-in annotations using XSD Type Representation in the XTypes specification, see the section “XSD Type Representation” in the OMG 'Extensible and Dynamic Topic Types for DDS' specification, version 1.3.
Chapter 6 TypeCode API Changes

As described in Type Representation (Chapter 5 on page 38), in Connext DDS 5.x and higher, only Modern C++ supports the DynamicType API described in the Extensible Types specification. For other language bindings, user applications can continue to use the TypeCode API to introspect the types at runtime.

The TypeCode API includes two operations to retrieve the extensibility kind of a type and the ID of a member:

- DDS_TypeCode_extensibility_kind()
- DDS_TypeCode_member_id()

The value of the following annotations currently cannot be accessed using the TypeCode API: @default, @default_literal, @range, @min, @max.

For information on these operations, see the API Reference HTML documentation (open ReadMe.html\(^1\) and select the API for your language, then select Modules, DDS API Reference, Topic Module, Type Code Support, DDS_TypeCode).

---

\(^1\)After installing Connext DDS, you will find ReadMe.html in the ndds.<version> directory.
Chapter 7 DynamicData API

Connext DDS 5.x and higher does not currently support the DynamicData API described in the Extensible Types specification. User applications should continue using the traditional DynamicData API.

The traditional DynamicData API has been extended to support optional members (see 3.2.2.4 Using Optional Member with DynamicData on page 27).

The traditional API does not currently support setting/getting the value of a DynamicData sample using member IDs as defined in the Extensible types specification. The member values of the following types should be accessed using the member name:

- Unions
- Struct
- Valuetypes

Although it is possible to use the member_id field in the get/set operations provided by the DynamicData API, the meaning of the ID in the API is not compliant with the member ID described in the Extensible Types specification.

For example, in the Extensible Types specification, the members of a union are identified by both the case values associated with them and their member IDs. When using the DynamicData API to set/get the value of a union member, the member_id parameter in the APIs corresponds to the case value of the member instead of the member ID.
Chapter 8 ContentFilteredTopics

Writer-side filtering using the built-in filters (SQL and STRINGMATCH) is supported as long as the filter expression contains members that are present in both the DataReader’s type and the DataWriter’s type. For example, consider the following types:

DataWriter:

```csharp
struct MyBaseType {
 int32 x;
};
```

DataReader:

```csharp
struct MyDerivedType : MyBaseType {
 public int32 y;
};
```

If the DataReader creates a ContentFilteredTopic with the expression “x>5”, the DataWriter will perform writer-side filtering since it knows how to find x in the outgoing samples.

If the DataReader creates a ContentFilteredTopic with the expression “x>5 and y>5” the DataWriter will not do writer side filtering since it does not know anything about “y”. Also, when the DataWriter tries to compile the filter expression from the DataReader, it will report an error such as the following:

```
DDS_TypeCode_dereference_member_name:member starting with [y > ] not found
PREParticipant_createContentFilteredTopicPolicy:content filter compile error 1
```

To learn how to use optional members in filter expressions, see 3.2.2.5 Using Optional Members in SQL Filter Expressions on page 28.
Chapter 9 RTI Spy

RTI Spy, rtiddsspy, includes limited support for Extensible Types:

- *rtiddsspy* will automatically create a *DataReader* for each version of a type discovered for a topic. In Connext DDS 5.x and higher, it is not possible to associate more than one type to a topic within a single *DomainParticipant*, therefore each version of a type will require its own *DomainParticipant*.

- The TypeConsistencyEnforcementQosPolicy’s *kind* in each of the *DataReaders* created by *rtiddsspy* is set to DISALLOW_TYPE_COERCION. This way, a *DataReader* will only receive samples from *DataWriters* with the same type, without doing any conversion.

- The *-printSample* option will print each of the samples using the type version of the original publisher.

For example:

```c
struct A {
    int32 x;
};
struct B {
    int32 x;
    int32 y;
};
```

Let’s assume that we have two *DataWriters* of *Topic* “T” publishing type “A” and type “B” and sending TypeObject information. After we start *Spy*, we will see output like this:

```
NddsSpy is listening for data, press CTRL+C to stop it.
source_timestamp    Info    Src HostId    topic          type
--------------------    ------    -----    -----------    ------
1345847910.453969    W +N    0A1E01C0    Example A    A
1345847912.056410    W +N    0A1E01C0    Example B    B
1345847914.454385    d +N    0A1E01C0    Example A    A
    x: 1
1345847916.056787    d +N    0A1E01C0    Example B    B
    x: 2
    y: 3
```
9.1 Type Version Discrimination

Rtiddsspy uses the rules described in 2.3.1 Rules For Type-Consistency Enforcement on page 13 to decide whether or not to create a new DataReader when it discovers a DataWriter for a topic “T”.

For DataWriters created with previous Connext DDS releases (4.5f and lower), rtiddsspy will select the first DataReader with a registered type name equal to the discovered registered type name, since DataWriters created with previous releases do not send TypeObject information.
Chapter 10 Compatibility with Previous Releases

For important information about compatibility issues when communicating with applications using an older (5.x) version of Connext DDS, please see the following documentation:

- If you are upgrading to 6.1.0 from a release older than 5.3.1, please first see this chapter in the Connext DDS Core Libraries Getting Started Guide Addendum for Extensible Types for 5.3.1. Then see the Migration Guide on the RTI Community Portal (https://community.rti.com/documentation) for migration issues related to upgrading from 5.3.1 to 6.1.0.

- If you are upgrading to 6.1.0 from 5.3.1, please see the Migration Guide on the RTI Community Portal (https://community.rti.com/documentation).