

RTI Connex DDS

Core Libraries and Utilities

Getting Started Guide

Addendum for Extensible Types

Version 5.1.0



Your systems. Working as one.



© 2012-2013 Real-Time Innovations, Inc.
All rights reserved.
Printed in U.S.A. First printing.
December 2013.

Trademarks

Real-Time Innovations, RTI, DataBus, and Connex are trademarks or registered trademarks of Real-Time Innovations, Inc. All other trademarks used in this document are the property of 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.

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

1	Introduction	1
2	Type Safety and System Evolution.....	2
2.1	Defining Extensible Types	3
2.2	Verifying Type Consistency: Type Assignability	4
2.3	Type-Consistency Enforcement	7
2.3.1	Rules For Type-Consistency Enforcement	7
2.4	Notification of Inconsistencies: INCONSISTENT_TOPIC Status	8
2.5	Built-in Topics.....	8
3	Type System Enhancements	9
3.1	Structure Inheritance	9
3.2	Optional Members	10
3.2.1	Defining Optional Members	10
3.2.2	Using Optional Members in an Application	10
4	Data Representation	16
5	Type Representation.....	16
5.1	XML and XSD Type Representations	17

6	TypeCode API Changes	17
7	TypeObject Resource Limits.....	18
8	DynamicData API.....	18
9	ContentFilteredTopics.....	19
10	Annotations.....	19
11	RTI Spy	20
	11.1 Type Version Discrimination	21
12	Compatibility with Previous Connex Releases	21
	12.1 Type-Consistency Enforcement	21
	12.2 Trigger for Changes to INCONSISTENT_TOPIC Status.....	22
	12.3 Wire Compatibility	22

Getting Started with Extensible Types

1 Introduction

This release of *RTI Connext* includes partial support for the "Extensible and Dynamic Topic Types for DDS" (DDS-XTypes) specification¹ from the Object Management Group (OMG). 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 now supported:

- ❑ Type definitions are now checked as part of the *Connext* 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.
- ❑ The above features are supported in the RTI core middleware in the C, C++, Java, and .NET² programming languages.

The following Extensible Types features are not supported:

- ❑ Optional members in .NET
- ❑ These types: BitSet, Map
- ❑ These built-in annotations: BitSet, Verbatim, MustUnderstand
- ❑ Annotation definition
- ❑ Standard syntax to apply annotations (see [Section 10](#))
- ❑ XML data representation (XML *type* representation is supported)
- ❑ Dynamic language binding compliant with the Extensible Types specification: Dynamic-Type and DynamicData (see [DynamicData API \(Section 8\)](#)).

1. <http://www.omg.org/spec/DDS-XTypes/>

2. Optional members are not supported in the .NET API.

- ❑ `DataRepresentationQosPolicy`
- ❑ type member in `PublicationBuiltinTopicData` and `SubscriptionBuiltinTopicData`
- ❑ Association of a topic to multiple types within a single `DomainParticipant`
- ❑ The 'null' keyword in SQL filter expressions

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 RTI's Downloads page (www.rti.com/downloads) or the RTI Support Portal (<https://support.rti.com/>). The portal requires an account name and password. 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.

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 extensible and mutable types. A type may be final, 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.
- ❑ **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:

```
struct A {
    long a; //@ID 10
    long b; //@ID 20
    long c; //@ID 30
}
and
struct B {
    long b; //@ID 20
    long a; //@ID 10
    long x; //@ID 40
}
```

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 data bus 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 data bus will compare the types it finds with the contracts you specified.

2.1 Defining Extensible Types

A type's kind of extensibility is applied with the **Extensibility** annotations seen in [Table 2.1](#). If you do not specify any particular extensibility, the default is extensible.

Table 2.1 Extensibility Annotations

IDL	<pre> struct MyFinalType { long x; }; //@Extensibility FINAL_EXTENSIBILITY struct MyExtensibleType { long x; }; //@Extensibility EXTENSIBLE_EXTENSIBILITY struct MyMutableType { long x; }; //@Extensibility MUTABLE_EXTENSIBILITY </pre>
XML	<pre> <struct name="MyFinalType" extensibility="final"> <member name="x" type="long"/> </struct> <struct name="MyExtensibleType" extensibility="extensible"> <member name="x" type="long"/> </struct> <struct name="MyMutableType" extensibility="mutable"> <member name="x" type="long"/> </struct> </pre>

Table 2.1 Extensibility Annotations

XSD	<pre> <xsd:complexType name="MyFinalType"> <xsd:sequence> <xsd:element name="x" minOccurs="1" maxOccurs="1" type="xsd:int"/> </xsd:sequence> </xsd:complexType> <!-- @struct true --> <!-- @extensibility FINAL_EXTENSIBILITY --> <xsd:complexType name="MyExtensibleType"> <xsd:sequence> <xsd:element name="x" minOccurs="1" maxOccurs="1" type="xsd:int"/> </xsd:sequence> </xsd:complexType> <!-- @struct true --> <!-- @extensibility EXTENSIBLE_EXTENSIBILITY --> <xsd:complexType name="MyMutableType"> <xsd:sequence> <xsd:element name="x" minOccurs="1" maxOccurs="1" type="xsd:int"/> </xsd:sequence> </xsd:complexType> <!-- @struct true --> <!-- @extensibility MUTABLE_EXTENSIBILITY --> </pre>
-----	--

Member IDs are set using the optional "@ID" annotation. For example:

```

struct MyType {
    long x; //@ID 10
    long y; //@ID 20
};

```

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.

```

struct MyType {
    long a;
    long b;
    long c; //@ID 100
    long 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 \(Section 4\)](#) for additional details).

Note: To specify both the "@ID" and "@Optional" annotations, they must be on separate lines:

```

struct MyType {
    long a;
    long b;
    long c; //@Optional
           //@ID 100
    long d;
};

```

2.2 Verifying Type Consistency: Type Assignability

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

In previous *Connex* releases, 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, *Connex* 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, *Connex* 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 [Type-Consistency Enforcement \(Section 2.3\)](#).)

For example, in [Table 2.2](#) the middleware can assign `MyMutableType1` to or from `MyMutableType2`, but not to or from `MyMutableType3`.

Table 2.2 **Mutable Types Example 1**

<pre>struct MyMutableType1 { long x; long y; } // @Extensibility MUTABLE_EXTENSIBILITY</pre> <p>Note: If you do not explicitly declare member IDs, they are assigned automatically starting with 0.</p>	<pre>struct MyMutableType2 { long y; // @ID 1 long z; // @ID 2 long x; // @ID 0 } // @Extensibility MUTABLE_EXTENSIBILITY</pre> <p><code>MyMutableType1</code> and <code>MyMutableType2</code> can be assigned to each other.</p>	<pre>struct MyMutableType3 { long y; long z; // @key long x; } // @Extensibility MUTABLE_EXTENSIBILITY</pre> <p><code>MyMutableType3</code> has two issues:</p> <ul style="list-style-type: none"> • The member IDs <code>x</code> and <code>y</code> do not match those of <code>MyMutableType1</code>. For example, the member ID of <code>x</code> is 0 in <code>MyMutableType1</code> but 2 in <code>MyMutableType3</code>. • <code>MyMutableType3</code> has an extra key member (<code>z</code>).
---	---	---

The type of a member in a mutable type can also change if the new type is assignable. For example, in [Table 2.3](#), `MyMutableType4` is assignable to or from `MyMutableType5` but not to or from `MyMutableType6`.

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.

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 [Optional Members \(Section 3.2\)](#)) 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.

Table 2.3 **Mutable Types Example 2**

<pre>struct NestedMutableType1 { long a; //@ID 10 }; //@Extensibility MUTABLE_EXTENSIBILITY struct NestedExtensibleType1 { string text; }; struct MyMutableType4 { NestedMutableType1 m1; NestedExtensibleType1 m2; } //@Extensibility MUTABLE_EXTENSIBILITY</pre>	<pre>struct NestedMutableType2 { short b; //@ID 20 long a; //@ID 10 }; //@Extensibility MUTABLE_EXTENSIBILITY struct NestedExtensibleType2 { string text; string title; }; struct MyMutableType5 { NestedMutableType2 m1; NestedExtensibleType2 m2; } //@Extensibility MUTABLE_EXTENSIBILITY MyMutableType5 and MyMutableType4 are assignable because the types of m1 and m2 are assignable too.</pre>	<pre>struct NestedMutableType3 { short b; //@ID 20 short a; //@ID 10 }; //@Extensibility MUTABLE_EXTENSIBILITY struct NestedExtensibleType3 { string title; string text; }; struct MyMutableType6 { NestedMutableType3 m1; NestedExtensibleType3 m2; } //@Extensibility MUTABLE_EXTENSIBILITY MyMutableType6 and MyMutableType4 are not assignable because the types of m1 and m2 are not assignable: <ul style="list-style-type: none"> • NestedExtensibleType3 is just extensible but adds a new member at the beginning • NestedMutableType3 changes the type of 'a' but the new type (short) is not assignable to the original type (long). </pre>
--	---	--

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 [Type-Consistency Enforcement \(Section 2.3\)](#)). 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 DDS-XTypes specification¹.

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 [Type-Consistency Enforcement \(Section 2.3\)](#)).

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 [TypeObject Resource Limits \(Section 7\)](#)).

1. <http://www.omg.org/spec/DDS-XTypes/>

If the logging verbosity of is set to `NDDS_CONFIG_LOG_VERBOSITY_WARNING` or higher, *Connex* 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 `TypeConsistencyEnforcementQosPolicy` defines the rules that determine whether the type used to publish a given data stream is consistent with that used to subscribe to it.

The `QosPolicy` structure includes the member in [Table 2.4](#).

Table 2.4 **DDS_TypeConsistencyEnforcementQosPolicy**

Type	Field Name	Description
DDS_TypeConsistencyKind	kind	Can be either: <ul style="list-style-type: none"> • <code>DISALLOW_TYPE_COERCION</code>: The <i>DataWriter</i> and <i>DataReader</i> must support the same data type in order for them to communicate. • <code>ALLOW_TYPE_COERCION</code>: The <i>DataWriter</i> and <i>DataReader</i> need not support the same data type in order for them to communicate, as long as the reader's type is assignable from the writer's type. (Default)

This `QosPolicy` defines a type consistency **kind**, which allows applications to choose either of these behaviors:

- ❑ `DISALLOW_TYPE_COERCION`: The *DataWriter* and *DataReader* must support the same data type in order for them to communicate. (This is the degree of type consistency enforcement required by the DDS specification prior to the Extensible Types 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 [Section 2.2](#).

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 `ALLOW_TYPE_COERCION`. However, when the middleware is introspecting the built-in topic data declaration of a remote *DataWriter* or *DataReader* in order to determine whether it can match with a local *DataReader* or *DataWriter*, if it observes that no `TypeConsistencyEnforcementQosPolicy` value is provided (as would be the case when communicating with a *Connex* implementation not in conformance with this specification), the middleware assumes a kind of `DISALLOW_TYPE_COERCION`.

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. If the *DataReader* does not allow type coercion, then its type must be structurally identical 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 *Connex* releases prior to 5.0.

If either Step 1 or Step 2 fails, the *Topics* associated with the *DataReader* and *DataWriter* are considered to be inconsistent (see [Section 2.4](#)).

The properties in [Table 2.5](#) 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 2.5 **Type Assignability Properties**

Property Name	Description
dds.type_consistency.ignore_member_names	<p>When set to 1, members of a type can change their name while keeping their member ID.</p> <p>For example, <i>MyType</i> and <i>MyTypeSpanish</i> are only assignable if this property is set to 1:</p> <pre>struct MyType { long x; //@ID 10 long angle; //@ID 20 }; struct MyTypeSpanish { long x; //@ID 10 long angulo; //@ID 20 };</pre> <p>This property also lets enumeration constants change their name while keeping their value.</p>
dds.type_consistency.ignore_sequence_bounds	<p>When set to 1, sequences and strings in a <i>DataReader</i> type can have a maximum length smaller than that of the <i>DataWriter</i> type. When the length of the sequence in a particular samples is larger than the maximum length, that sample is discarded.</p>

2.4 Notification of Inconsistencies: INCONSISTENT_TOPIC Status

Every time a *DataReader* and a *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 *Connex* releases where the change of status occurred when a remote *Topic* inconsistent with the locally created *Topic* was discovered.

2.5 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 2.6](#)).

Table 2.6 **New Field in Subscription Builtin Topic Data**

Type	New Field	Description
DDS_TypeConsistencyEnforcementQosPolicy	type_consistency	Indicates the type_consistency requirements of the remote <i>DataReader</i> (see Section 2.3).

You can retrieve this information by subscribing to the built-in topics and using the *DataReader*'s `get_matched_publication_data()` operations.

3 Type System Enhancements

3.1 Structure Inheritance

A structure can define a base type as seen in [Table 3.1](#). 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

IDL	<pre>struct MyBaseType { long x; }; struct MyDerivedType : MyBaseType { long y; };</pre>
XML	<pre><struct name="MyBaseType"> <member name="x" type="long"/> </struct> <struct name=" MyDerivedType" baseType="MyBaseType"> <member name="y" type="long"/> </struct></pre>
XSD	<pre><xsd:complexType name="MyBaseType"> <xsd:sequence> <xsd:element name="x" minOccurs="1" maxOccurs="1" type="xsd:int"/> </xsd:sequence> </xsd:complexType> <!-- @struct true --> <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 --></pre>

In *Connex* 5.0 and higher, value types are equivalent to structures. You can still use value types, but support for this feature may be discontinued in future releases.

For example:

```
struct MyType {
    long x;
};

valuetype MyType {
    public long 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 valuetype'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 \(Section 2\)](#)), optional members can be explicitly unset.

Using optional members in your types can be useful if you want to reduce bandwidth usage—*Connex*t 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](#)). 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.

Note: To specify both the “@ID” and “@Optional” annotations, they must be on separate lines:

```
struct MyType {
    long a;
    long b;
    long c; //@Optional
           //@ID 100
    long d;
};
```

Table 3.2 Declaring Optional Members

IDL	<pre>struct MyType { long optional_member; //@Optional long non_optional_member; };</pre>
XML	<pre><struct name="MyType"> <member name="optional_member" optional="true" type="long"/> <member name="non_optional_member" type="long"/> </struct></pre>
XSD	<pre><xsd:complexType name="MyType"> <xsd:sequence> <xsd:element name="optional_member" minOccurs="0" maxOccurs="1" type="xsd:int"/> <xsd:element name="non_optional_member" minOccurs="1" maxOccurs="1" type="xsd:int"/> </xsd:sequence> </xsd:complexType> <!-- @struct true --></pre>

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/C++

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:

```
struct MyType {
    long optional_member1; //@Optional
    Foo optional_member2; //@Optional
    long non_optional_member;
};
```

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

```
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	<pre>Foo* FooTypeSupport_create_data_w_params(const struct DDS_TypeAllocationParams_t* alloc_params) DDS_ReturnCode_t FooTypeSupport_delete_data_w_params(struct Foo* a_data, const struct DDS_TypeDeallocationParams_t* dealloc_params)</pre>
C++	<pre>Foo* FooTypeSupport::create_data(const DDS_TypeAllocationParams_t& alloc_params); DDS_ReturnCode_t FooTypeSupport::delete_data(TData* a_data, const DDS_TypeDeallocationParams_t& dealloc_params);</pre>

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 members, use the following functions:

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

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):

```
// Create and send a sample where all optional members are set
MyType * data = MyTypeTypeSupport::create_data(
    DDS_TypeAllocationParams_t().set_allocate_optional_members(
        DDS_BOOLEAN_TRUE));

*data->optional_member1 = 1;
strcpy(data->optional_member2->text, "hello");
data->non_optional_member = 2;
writer->write(*data, DDS_HANDLE_NIL);
```

```

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

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

// Create and send a sample where all optional members are unset
data = MyTypeTypeSupport::create_data();

data->non_optional_member = 3;
writer->write(*data, DDS_HANDLE_NIL);

// Now send optional_member1:
data->optional_member1 = (DDS_Long *)
DDS_Heap_malloc(sizeof(DDS_Long));

*data->optional_member1 = 4;
writer->write(*data, DDS_HANDLE_NIL);

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

```

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

```

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

data = MyTypeTypeSupport::create_data();
if (data == NULL) { /* error */ }

// Read or take as usual
reader->take_next_sample(*data, info);

if (info.valid_data) {
    std::cout << "optional_member1 ";
    if (data->optional_member1 != NULL) {
        std::cout << "=" << *data->optional_member1 << "\n";
    } else {
        std::cout << "is not set\n";
    }
    std::cout << "non_optional_member = "
        << data->non_optional_member << "\n";
}
// Delete the sample
MyTypeTypeSupport::delete_data(data);

```

3.2.2.2 Using Optional Members in Java

Optional members have the same mapping to Java class members as non-optional members, except that **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:

```
struct MyType {
    long optional_member1; //@Optional
    Foo optional_member2; //@Optional
    long non_optional_member;
};
```

This type maps to this Java class:

```
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:

```
// 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_member1
data.optional_member1 = 4;
writer.write(data, InstanceHandle_t.HANDLE_NIL);
```

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

```
// 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) {
    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);
}

```

3.2.2.3 Using Optional Member with DynamicData

This version of *Connex* supports a pre-standard version of `DynamicData` (see [DynamicData API \(Section 8\)](#)). 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 `DynamicData` type support and a `DynamicData` topic. More information and examples are available in the API Reference HTML documentation (select **Modules, RTI Connex DDS API Reference, Topic Module, Dynamic Data**).

```

// 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
DDS_DynamicData optionalMember2(NULL, DDS_DYNAMIC_DATA_PROPERTY_DEFAULT);
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);

// Delete the sample
type_support.delete_data(data);

```

In this example we read samples that contain optional members:

```

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 << "\n";
    } 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 << "\n";
    } else {
        std::cout << "Error getting non_optional_member\n";
    }
}
// Delete the sample
type_support->delete_data(data);

```

3.2.2.4 Using Optional Members in SQL Filter Expressions

SQL filter expressions used in `ContentFilteredTopics` and `QueryConditions` (see [ContentFilteredTopics \(Section 9\)](#) in this document and Section 4.6.7 (`ReadConditions` and `QueryConditions`) and Section 5.4 (`ContentFilteredTopics`) in the *Core Libraries And Utilities Users Manual*) can refer to optional members. The syntax is the same as for any other member.

For example, given the type `MyType`:

```

struct Foo {
    string text;
};

struct MyType {
    long optional_member1; //@Optional
    Foo optional_member2; //@Optional
    long 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 (`=`, `<>`, `<`, or `>`) 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.

4 Data Representation

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

*Connex*t uses Extended CDR, which defines an extension of the OMG CDR representation that is able to accommodate both optional members and mutable types.

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

For mutable types and optional members, *Connex*t uses parameterized CDR representation, 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 16,218.

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 *Connex*t, 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 (short, unsigned short, long, unsigned long, long long, unsigned long long, float, double, long double, boolean, octet, char) use a 4-byte header.
- Otherwise, if the member is an enumeration use 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.

5 Type Representation

Earlier versions of *Connex*t (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.

*Connex*t 5.x supports TypeObjects as the wire representation. To maintain backward compatibility with previous releases, *Connex*t 5.x still supports propagation of TypeCodes. However, the support for this feature may be discontinued in future releases.

Important: Connext does not support TypeObjects for types containing bitfields or for sparse types.

Connex 5.x does not currently support the DynamicType API described in the Extensible Types specification. Therefore you must continue using the TypeCode API to introspect the types at run time.

You can introspect the discovered type independently of the wire format by using the `type_code` member in the `PublicationBuiltinTopicData` and `SubscriptionBuiltinTopicData` structures.

Important: 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:

	Connex 5.x	Connex 4.5f and Earlier
Wire Representation	TypeObjects or TypeCodes (for backwards compatibility)	TypeCodes
For Introspection at Run Time	TypeCode API (DynamicType API currently not supported)	TypeCode API
Maximum Size of Serialized Representation	When using TypeObjects: Unbounded When using TypeCodes: 65 KB	65 KB

5.1 XML and XSD Type Representations

The XML and XSD type-representation formats available in *Connex* formed the basis for the DDS-XTypes specification of these features. They support the new features introduced in *Connex* 5.0. However, they have not been completely updated to the new standard format.

6 TypeCode API Changes

As described in [Section 5](#), *Connex* 5.x does not currently support the DynamicType API described in the Extensible Types specification. User applications can continue using the TypeCode API to introspect the types at run time.

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()`

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 *Connex*, you will find `ReadMe.html` in the `ndds.<version>` directory.

7 TypeObject Resource Limits

Table 7.1 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 7.1 New `TypeObject` Fields in `DomainParticipantResourceLimitsQosPolicy`

Type	Field	Description
DDS_Long	<code>type_object_max_serialized_length</code>	The maximum length, in bytes, that the buffer to serialize a <code>TypeObject</code> can consume. This parameter limits the size of the <code>TypeObject</code> that a <code>DomainParticipant</code> is able to propagate. Since <code>TypeObjects</code> 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 <code>TypeObjects</code> larger than the default maximum of 3072 bytes. This field allows you to specify a larger value. It cannot be UNLIMITED. Default: 3072
DDS_Long	<code>type_object_max_deserialized_length</code>	The maximum number of bytes that a deserialized <code>TypeObject</code> can consume. This parameter limits the size of the <code>TypeObject</code> that a <code>DomainParticipant</code> is able to store. Default: UNLIMITED
DDS_Long	<code>deserialized_type_object_dynamic_allocation_threshold</code>	A threshold, in bytes, for dynamic memory allocation for the deserialized <code>TypeObject</code> . Above it, the memory for a <code>TypeObject</code> 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

The `TypeObject` is needed for type-assignability enforcement.

By default, `Connex` will propagate both the pre-standard `TypeCode` and the new standard `TypeObject`. It is also possible to send either or none of them:

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

8 DynamicData API

`Connex 5.x` 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 [Using Optional Member with `DynamicData`](#) (Section 3.2.2.3)).

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.

9 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:

```
struct MyBaseType {
    long x;
};
```

DataReader:

```
struct MyDerivedType : MyBaseType {
    public long 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
PRESParticipant_createContentFilteredTopicPolicy:content filter compile
error 1
```

To learn how to use optional members in filter expressions, see [Using Optional Members in SQL Filter Expressions \(Section 3.2.2.4\)](#).

10 Annotations

The standard syntax for applying annotations is not supported in this release. The old, pre-standard format used by RTI is still in use for the built-in annotations added in *Connex* 5.x.

For example, the following DDS-XTypes conformant structure:

```
@Extensibility(EXTENSIBLE_EXTENSIBILITY) @Nested
struct MyType {
```

```

    @Key long x;
    @Shared long y;
};

```

Can be defined using:

```

struct MyType {
    long x; //@Key
    long * y;
};
//@Extensibility EXTENSIBLE_EXTENSIBILITY
//@top-level false

```

And this:

```

enum MyEnum {
    @Value(10) CONSTANT_1,
    @Value(20) CONSTANT_2
};

```

Can be defined using:

```

enum MyEnum {
    CONSTANT_1 = 10,
    CONSTANT_2 = 20
};

```

11 RTI Spy

rtiddsspy includes limited support for Extensible Types:

- ❑ *rtiddsspy* can subscribe to topics associated with final and extensible types.
- ❑ *rtiddsspy* will automatically create a *DataReader* for each version of a type discovered for a topic. In Connex 5.x, 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** will print each of the samples using the type version of the original publisher.

For example:

```

struct A {
    long x;
};
struct B {
    long x;
    long 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 get an output like this:

```

RTI Data Distribution Service Spy built with NDDS version 1.6a.00--
C1.6a.00--C++1.6a.00
Copyright 2012 Real-Time Innovations, Inc.
~~~~~
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				
1345847918.455104	d +M	0A1E01C0	Example A	A
x: 2				
1345847920.057084	d +M	0A1E01C0	Example B	B
x: 4				
y: 6				

11.1 Type Version Discrimination

Rtiddsspy uses the rules described in [Section 2.3.1](#) to decide whether or not to create a new *DataReader* when it discovers a *DataWriter* for a topic "T".

For *DataWriters* created with previous *Connex* 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.

12 Compatibility with Previous Connex releases



This section describes important behavior differences between *Connex* 5.x and previous releases. Please read this section to learn about possible incompatibility issues when communicating with pre-*Connex* 5.x applications.

12.1 Type-Consistency Enforcement

By default, *Connex* 5.x determines if a *DataWriter* and a *DataReader* can communicate by comparing the structure of their topic types (see [Section 2.2](#) and [Section 2.3](#)).

In previous releases, *Connex* considered only the registered type names.

This change in default behavior may cause some applications that were communicating when using previous releases to not communicate when ported to *Connex* 5.x.

For example, let's assume that we have the following two applications:

- The first application creates a *DataWriter* of Topic, **Square**, with the registered type name, **ShapeType**, and the type, **EnglishShapeType**:

```
struct EnglishShapeType {
    long x;
    long y;
    long size;
    float angle;
};
```

- The second application creates a *DataReader* of Topic, **Square**, with the registered type name **ShapeType**, and the type **SpanishShapeType**:

```
struct SpanishShapeType {
    long x;
    long y;
    long tamagno;
    float angulo;
};
```

Before Connexr 5.0: The *DataWriter* and *DataReader* in the above example *will* communicate. *Connexr* only considers the registered type name to determine whether or not the types were consistent; therefore the *DataWriter* and *DataReader* in the above example match because they both use the same registered type name, **ShapeType**.

Starting with Connexr 5.0: The *DataWriter* and *DataReader* in the above example will *not* communicate. The "Extensible and Dynamic Topic Types for DDS" specification does not consider **EnglishShapeType** and **SpanishShapeType** to be compatible. *Types are incompatible if they have members with same member ID¹ but different names.* In this case, **size** and **tamagno** have the same ID but different names (same situation for **angle** and **angulo**).

To make these two applications compatible, you can enable the property **dds.type_consistency.ignore_member_names**.

For more details on type consistency and assignability, see [Section 2.2](#) and [Section 2.3](#).

In Connexr 5.1 and higher: In *Connexr* 5.0, the middleware did not take into account the maximum length of a sequence or a string to determine if two types are assignable. Starting with release 5.1, the maximum length sequence of the *DataReader* type must be greater than or equal to that of the *DataWriter*. Enabling the old behavior is still possible (see [Section 2.3](#)).

12.2 Trigger for Changes to INCONSISTENT_TOPIC Status

The condition under which the middleware triggers an INCONSISTENT_TOPIC status update is different starting in release 5.0.

Before Connexr 5.0: The change of status occurred when a remote *Topic* inconsistent with the locally created *Topic* was discovered. This check was based only on the registered type name.

Starting with Connexr 5.0: The change of status occurs when a *DataReader* and a *DataWriter* on the same *Topic* do not match because the type-consistency enforcement check fails.

12.3 Wire Compatibility

An endpoint (*DataWriter* or *DataReader*) created with *Connexr* 5.x will not be discovered by an application that uses a previous *Connexr* release (4.5f or lower) if that endpoint's *TypeObject* is sent on the wire *and* its size is greater than 65535 bytes.

This is because *TypeObjects* with a serialized size greater than 65535 bytes require extended parameterized encapsulation when they are sent as part of the endpoint discovery information. This parameterized encapsulation is not understood by previous *Connexr* releases.

1. For information on member IDs, see the XTypes specification (<http://www.omg.org/spec/DDS-XTypes/>).